Heterogeneous System-on-Chip (SoC): On-chip Interconnect

Daniel Müller-Gritschneder

16.05.2024

Motivation

- Most chips feature a range of processing elements (PEs) / multi-cores
- PEs needs to communicate with each other
- On-chip Interconnect architecture and type play crucial role in performance.
- Chips and devices are connected via different types of interconnects

Agenda

- Interconnect types
- On-chip buses
- Networks-on-chip (NoC)

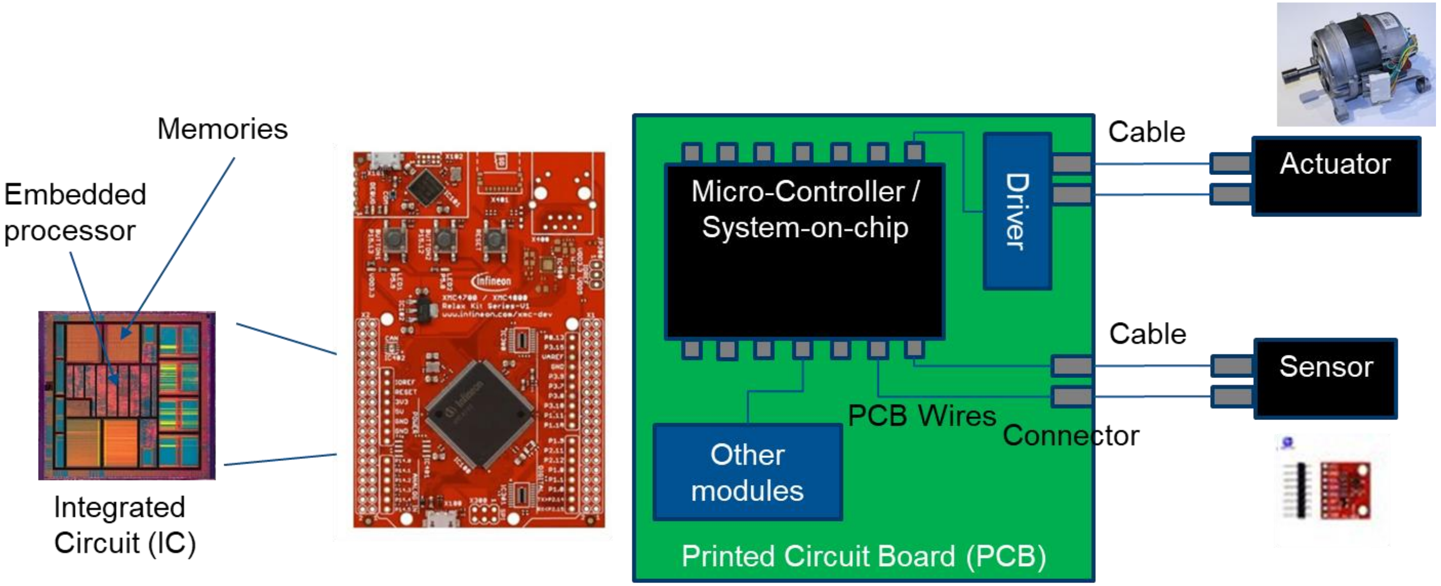
- A look at real on-chip interconnects

Optional, not relevant for exam

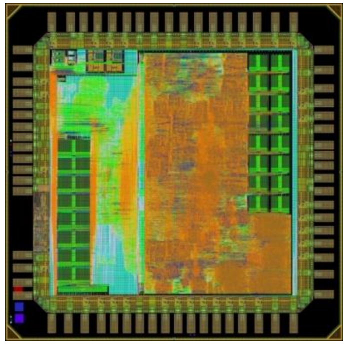
Interconnect Types

Interconnect Types

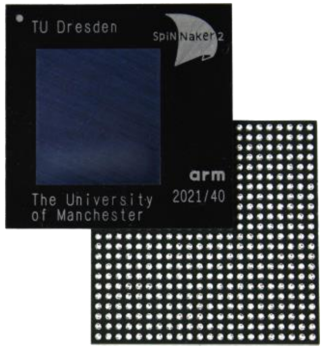
- On-Chip: Connects modules that are integrated into the same chip (IC: integrated circuit)
- PCB-level: Connects different ASICs + connectors and other component all mounted on one Printed Circuit Board (PCB).
- Many other interconnects (board to board, rack to rack): PCIe, Ethernet, CAN, UART, I2C, SPI, GPIO, …



Different Scales of Interconnects

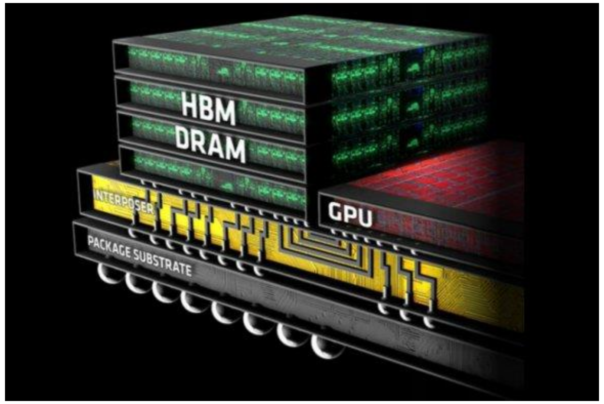


On-chip Interconnect

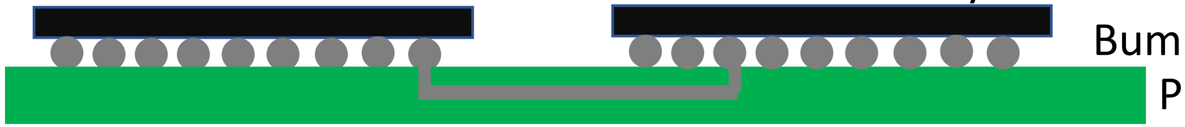


Chip Package

Chip2Chip (3D Stacked)



Source AMD



Processor

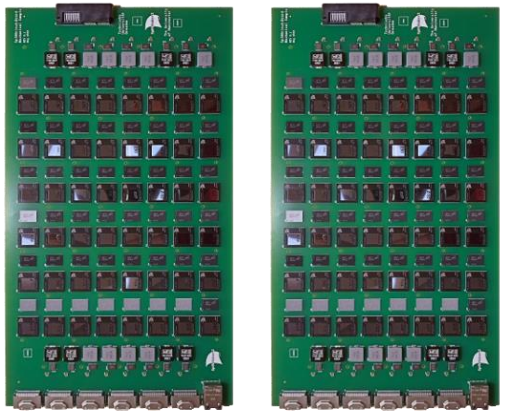
Main Memory

Bumps

PCB Board

Chip2Chip

Board2Board



Rack2Rack

Sources: Pulp, SpiNNCloud

On-chip Buses

Memory-mapped Buses

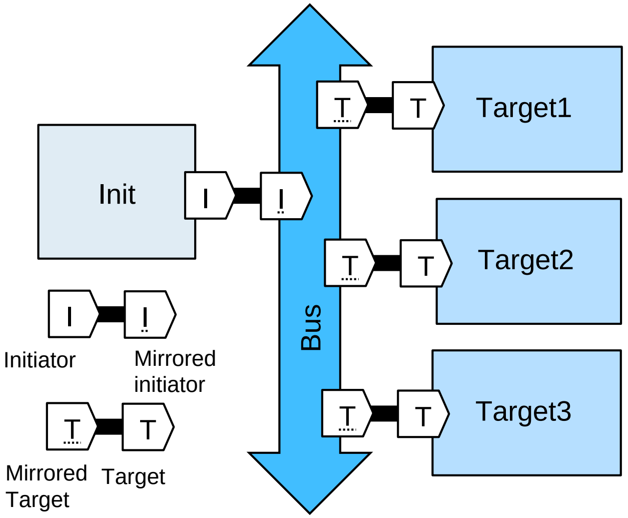
- Purpose:
  - Read or write a value from or to a certain address
  - Value can be data or peripheral control information
- Memory-mapped Bus has several (sub-)buses (group of signals) and a defined bus protocol
  - Address bus
  - Data bus for reading data
  - Data bus for writing data
  - Control signals: Indicate if access is read or write, bust length, ID, bus grant, …
- Modules on the bus can either act as initiators or targets
  - Typical initiators: CPUs, DSPs, DMAs, bus bridges, …
  - Typical targets: Memory, accelerators, interface peripheral, bus bridges, …

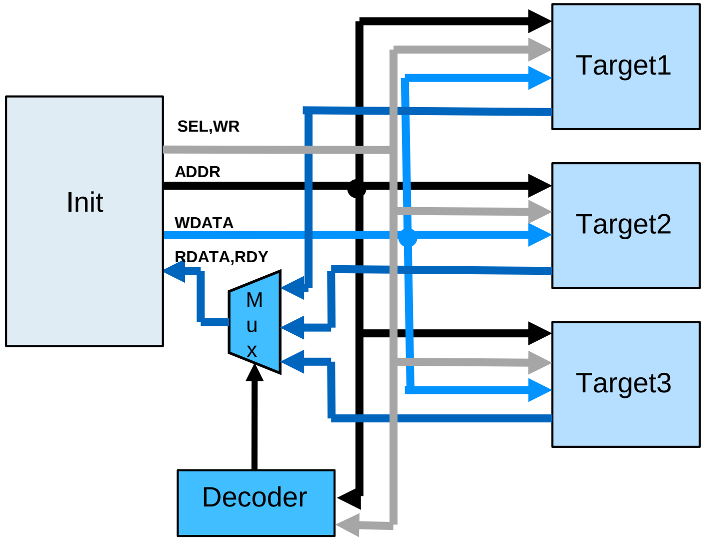
Classes of Memory-mapped Buses

- Single-initiator bus:
  - One initiator component can address different target components, which are mapped to different addresses
- Shared bus:
  - There are several initiators on the bus
  - An arbiter decides which initiator module is granted access to the bus
  - Only one initiator can access one slave via the bus at a time
- Layered bus:
  - There is more than one arbiter such that more than one initiator is granted access on the bus
  - Only one target component on each layer can be accessed at a time
- Crossbar/ bus matrix
  - Each target component has its own arbiter
  - Each target component can be accessed by one initiator at a time

Single-Initiator Bus

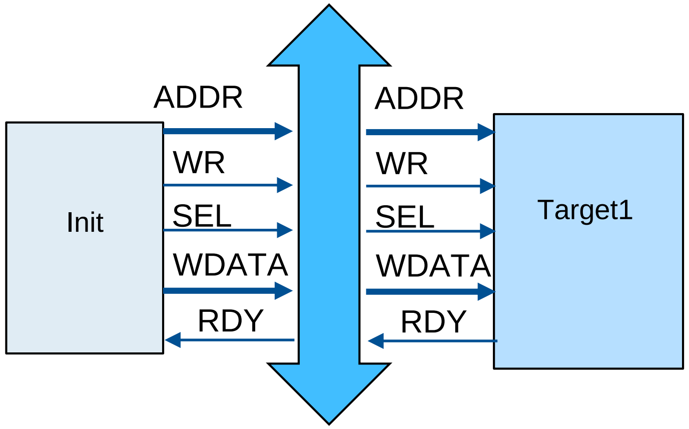
- Target knows
  - if it is addressed by observing the address bus ADDR
  - or decoder generates SEL signal for targets based on address bus ADDR
- Target can receive data on write data bus WDATA
- Decoder forwards the data from the addressed target by multiplexing it to the read data bus RDATA
- Additional control bus CTRL for signals related to bus protocol (e.g. WR, SEL, RDY)

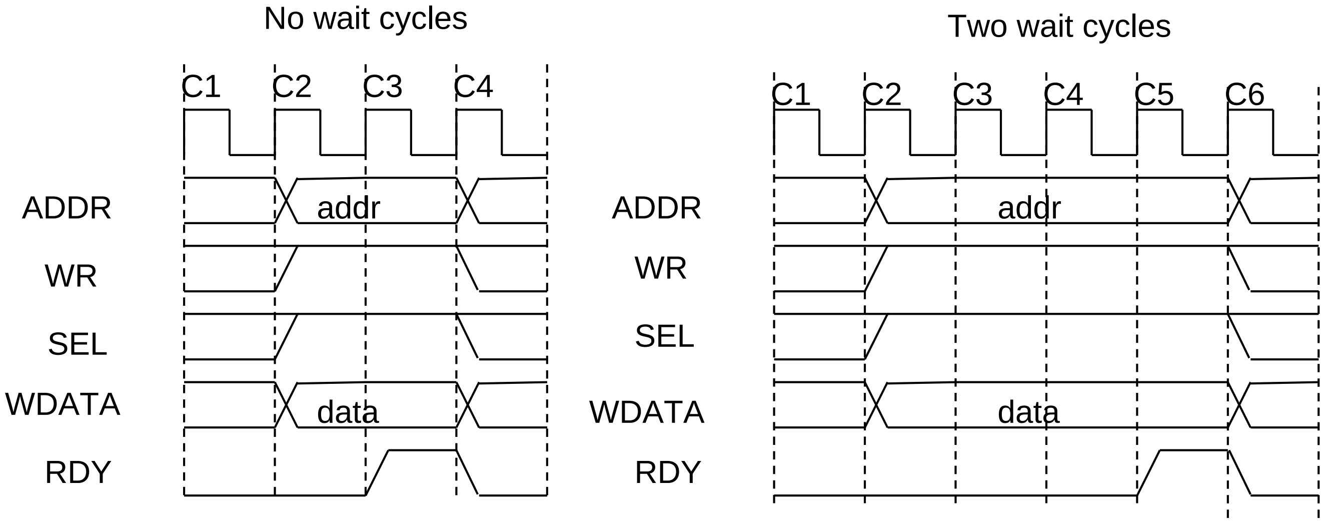




Simple Write Access

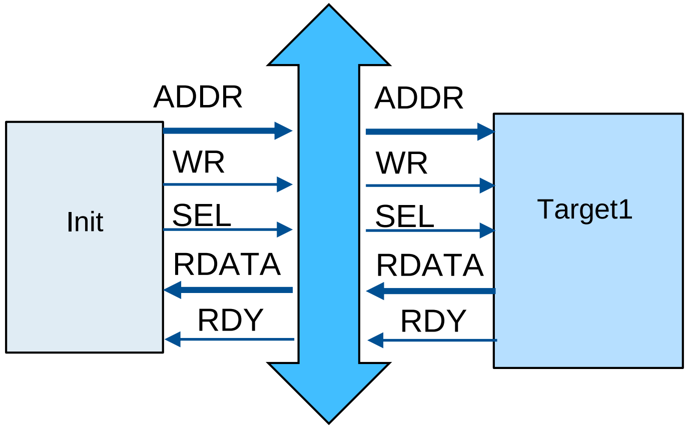
1. Initiator places address and data on the ADDR and WDATA bus
   Initiator indicates write by setting signal WR to high
   Initiator indicates that access is started by setting SEL signal to high
2. Target acknowledges write access by RDY signal

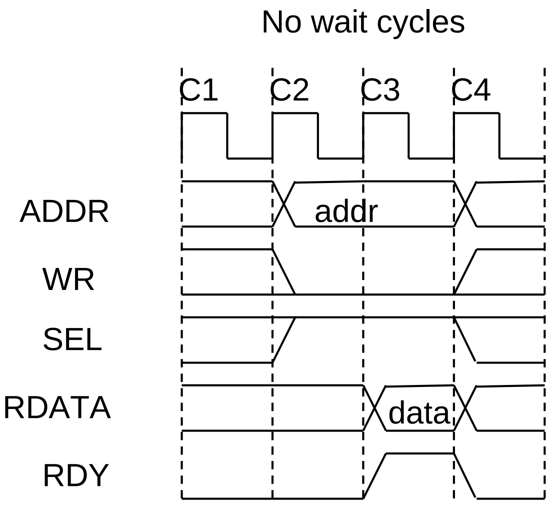


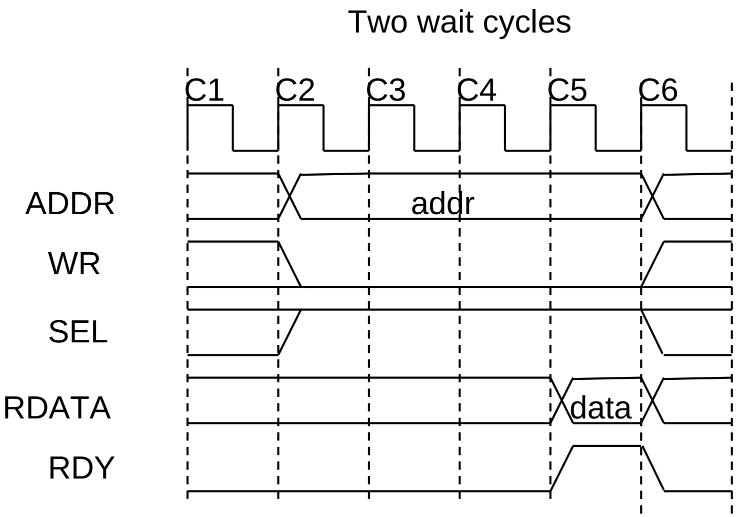


Simple Read Access

1. Initiator places address on the ADDR bus
   Initiator indicates read access by setting signal WR to low
   Initiator indicates that access is started by setting SEL signal to high
2. Target places data on RDATA bus
   Target acknowledges write access by RDY signal



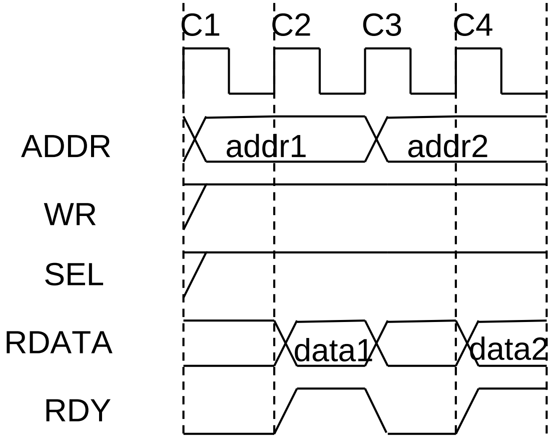




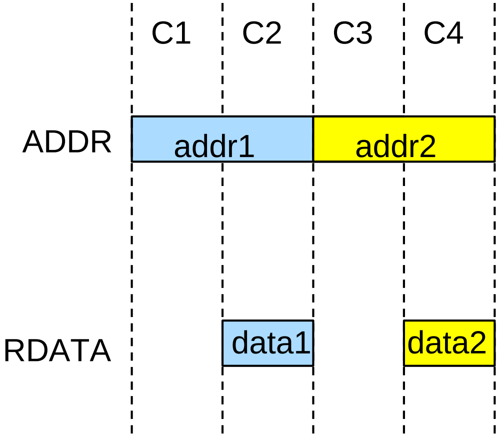
Performance of Simple Accesses

- Each access takes minimally two cycles
- Maximal bus bandwidth is: $BW_{bus} = 0.5 \cdot buswidth \cdot f_{bus}$

Two read accesses



Two read accesses (bus access diagram)



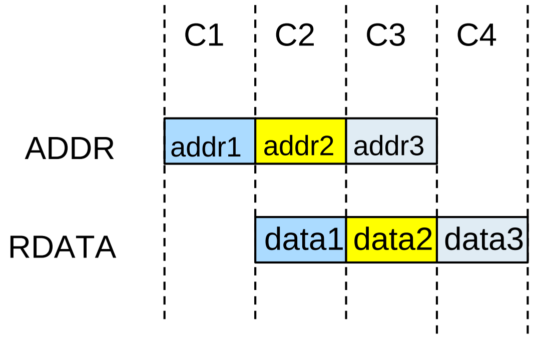
Pipelined Accesses

- The next address can be placed on the bus while the data is read
- Maximal bandwidth supported by bus is equal to:

$$BW_{bus} = buswidth \cdot f_{bus}$$

- Additional control signals and logic required to support pipelined accesses.

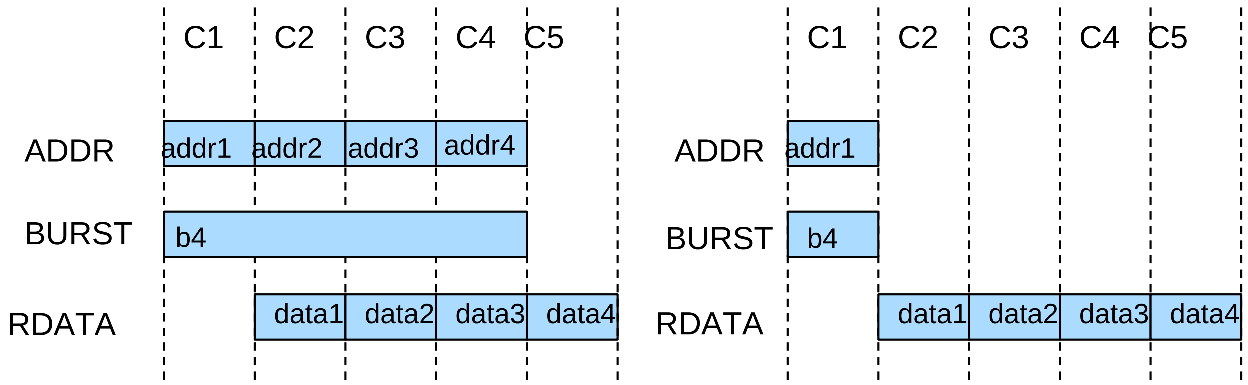
Three pipelined read accesses



Burst Accesses

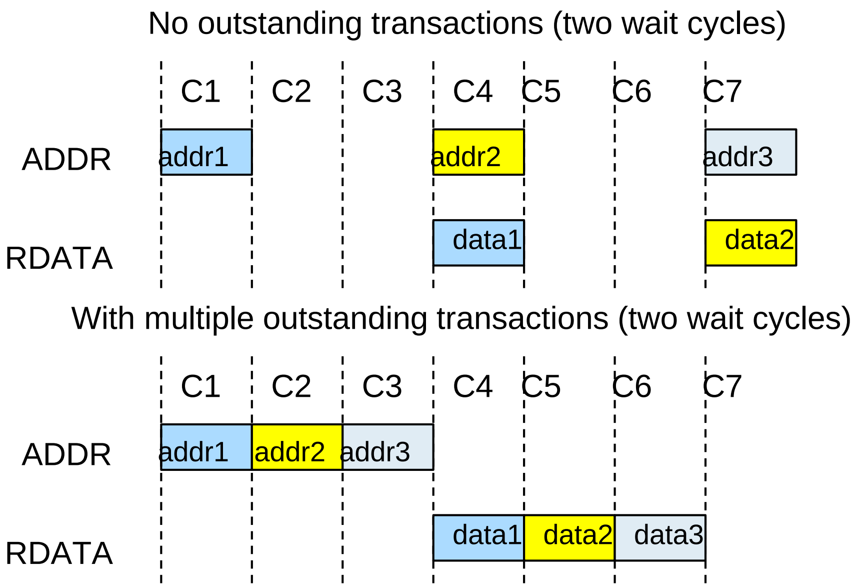
- A burst accesses a consecutive row of addresses
- Version 1: the addresses for all accesses must be given and a control signal that indicates that this is a burst access of a certain size
- Version 2: Only the start address must be given and a control signal that indicates that this is a burst access of a certain size

Four data values are returned for one start address (burst4)



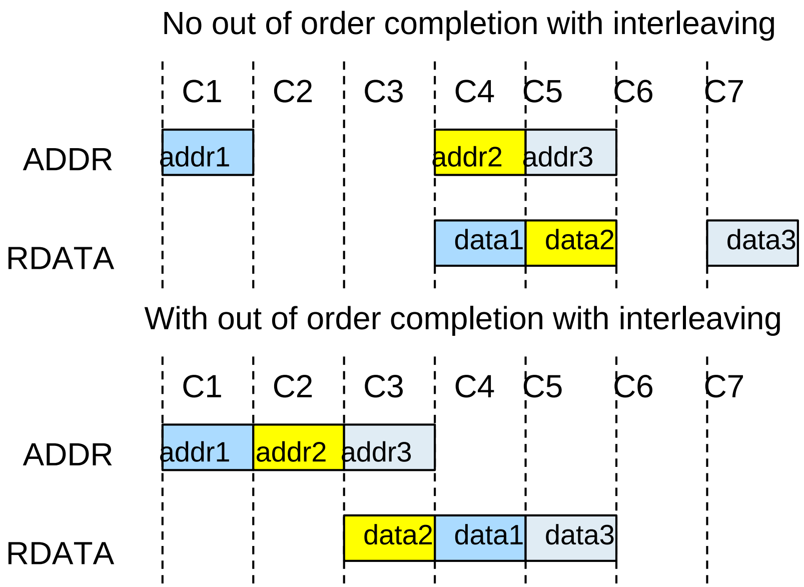
Multiple Outstanding Transactions

- A address may be placed on the bus before the data of the previous access has been read or be written
- This improves performance in case of wait cycles.


No outstanding transactions (two wait cycles)
C1
C2
C3
C4
C5
C6
C7
ADDR
addr1
addr2
addr3
RDATA
data1
data2
With multiple outstanding transactions (two wait cycles)
C1
C2
C3
C4
C5
C6
C7
ADDR
addr1
addr2
addr3
RDATA
data1
data2
data3

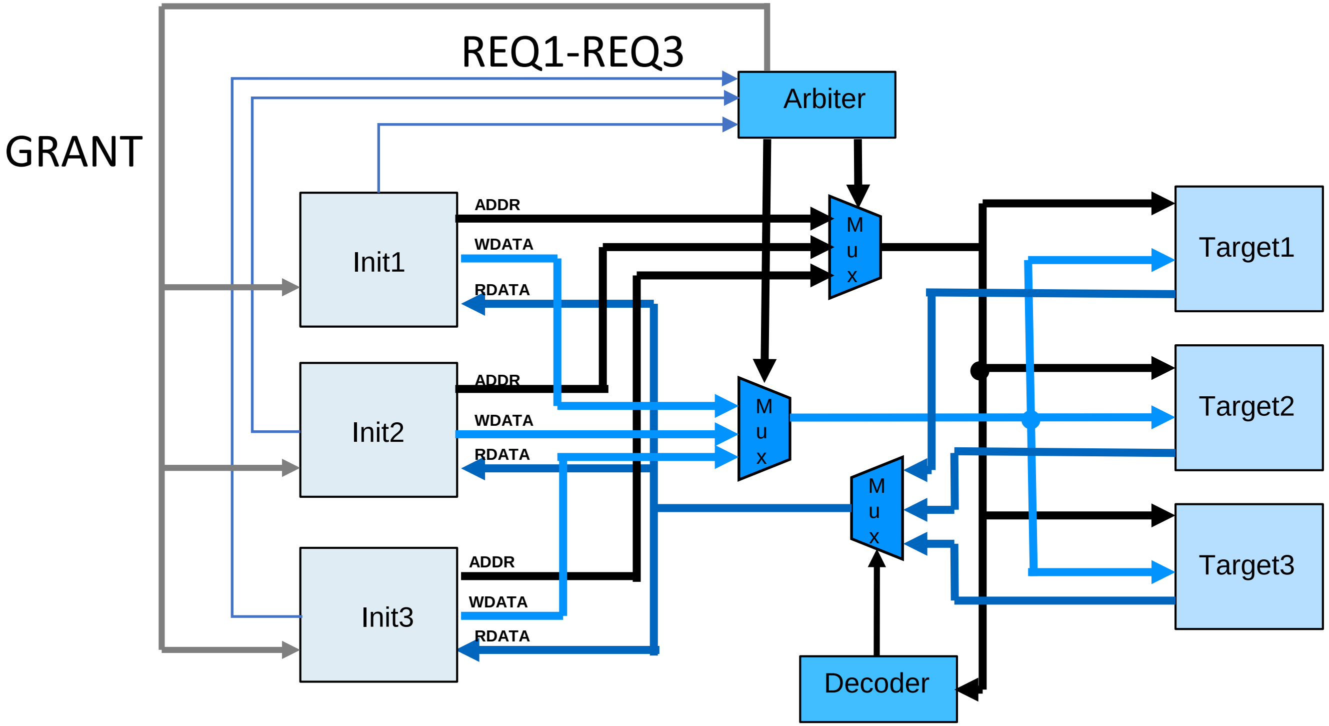
Out of order Completion with Interleaving

- A address may be placed on the bus before the data of the previous access has been read or be written
- In case of wait cycles, the order of data reads may be changed



Shared Bus

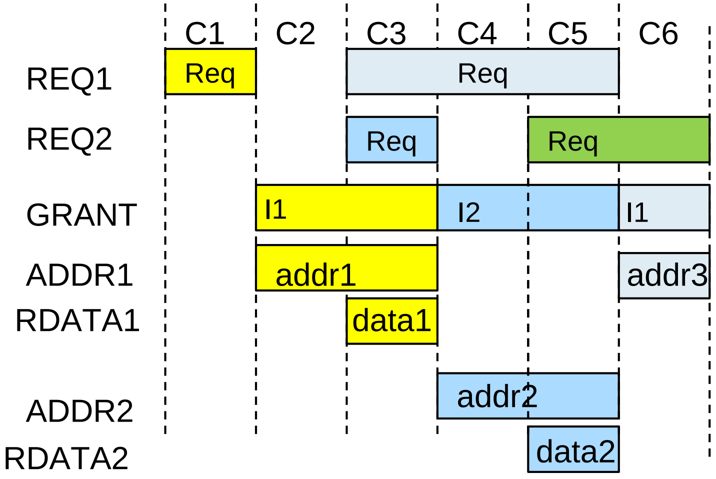
- Arbiter grants access to the initiator:
- Only the address and data of one initiator is forwarded to the targets



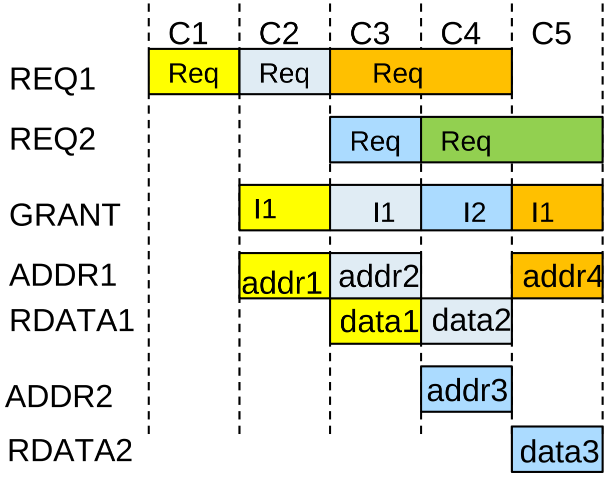
Bus Arbitration

- The arbiter grants access to initiator that request the bus
- Round-robin: Access granted to initiators in pre-defined order that is repeated
- FIFO: First initiator requesting the bus is granted access
- Priority: Initiator with highest priority is granted access to the bus

Round-robin: No pipelining

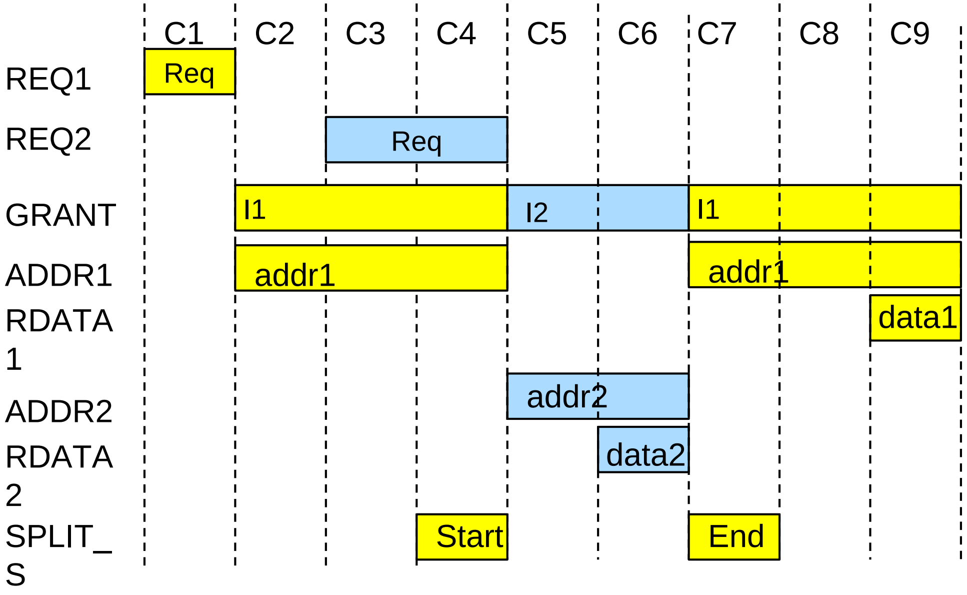


Round-robin: With pipelining



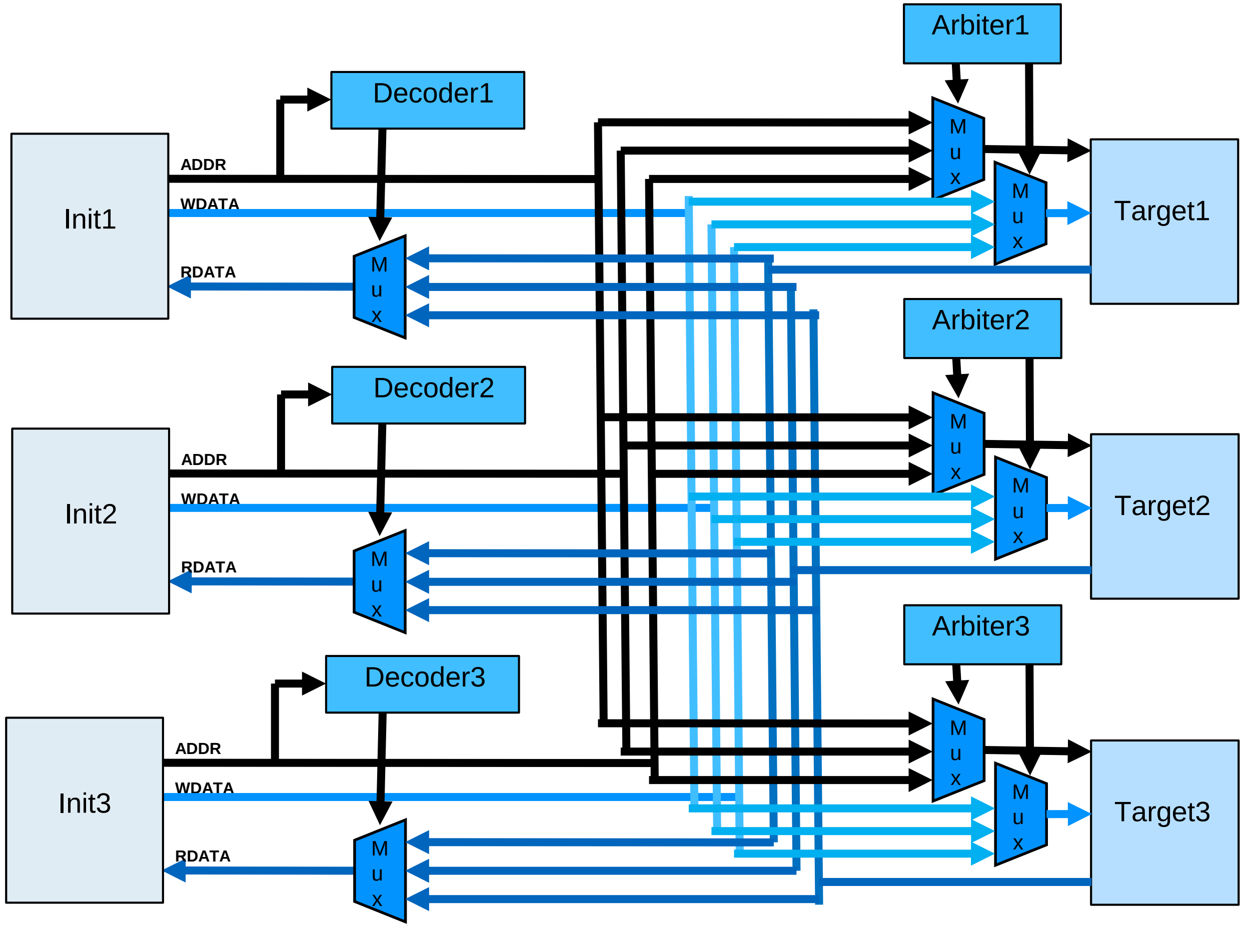
Split Accesses

- Slave can allow a split of an access if it was many wait cycles
- Access of initiator I1 is split by issuing a start of split by slave
- I2 is granted the bus and access of initiator I2 is performed
  Then access of initiator I1 is finished by issuing an end of split



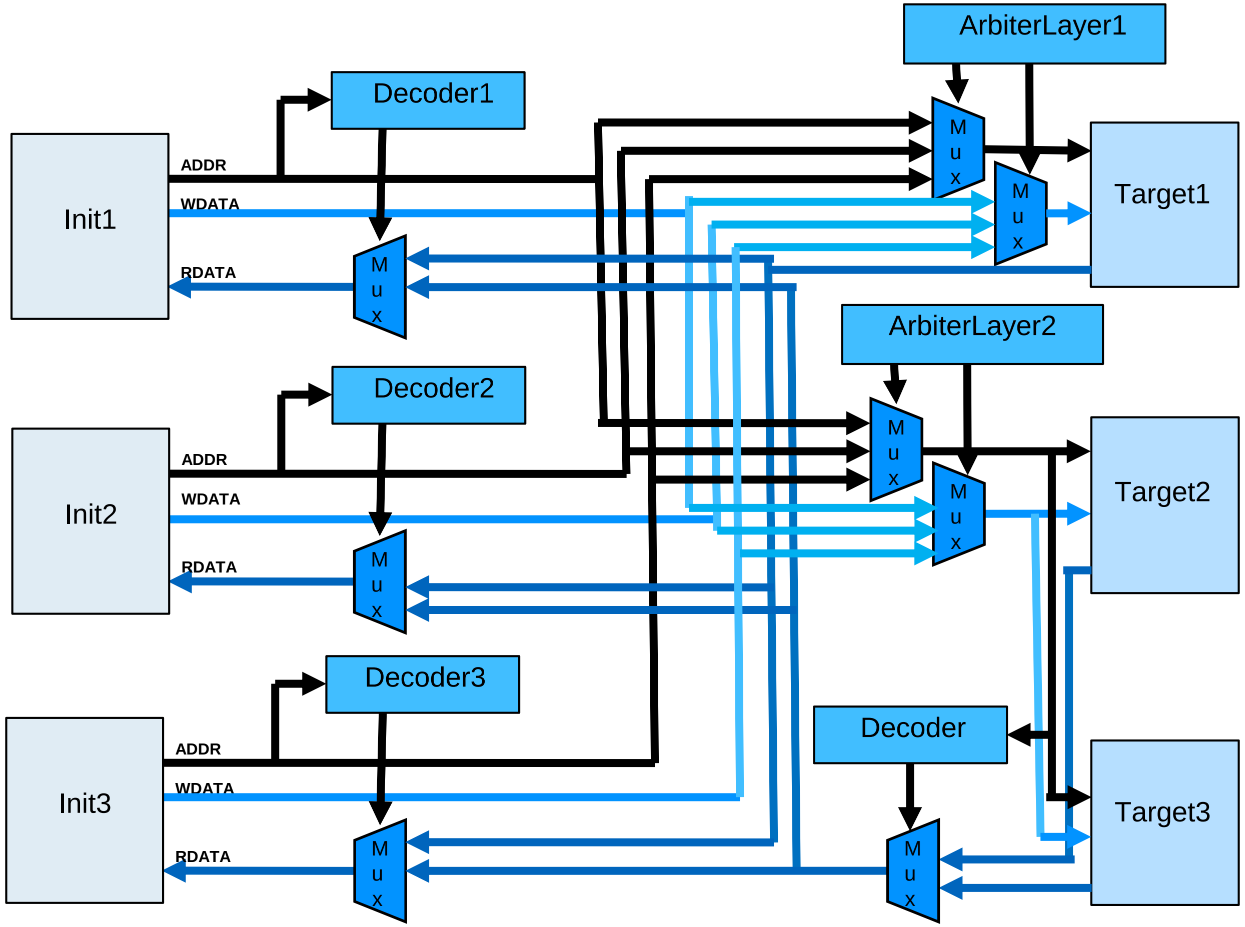
Crossbar / Bus Matrix

- All targets can be accessed individually
- Only conflict when two initiators access same target
- GRANT/REQ omitted.



Layered Bus

- Targets are on different layers
- Initiator can connect to targets on different layers simultaneously



Some Bus Standards

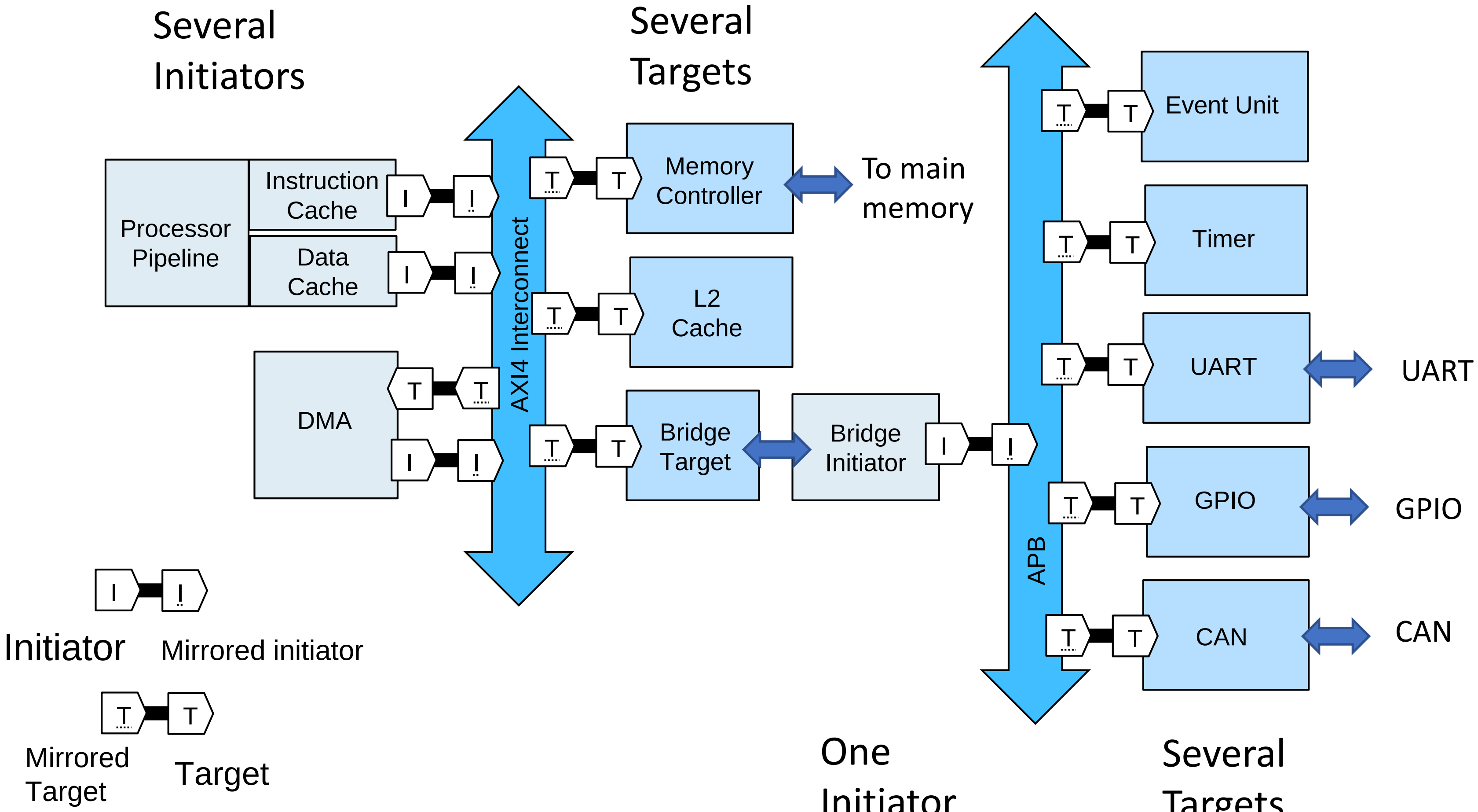
- AMBA Bus (ARM)
  - AHB: Advanced High Performance Bus
  - APB: Advanced Peripheral Bus
  - AXI: Advanced eXetendible Interface
- Wishbone (Open)
- TileLink (Open)

ARM AMBA Standard

- Different Versions e.g., AMBA 2,0, AMBA 3.0,…
- AHB: Advanced High Performance Bus
  - High performance
  - Pipelined operation
  - Multiple bus initiators
  - Burst transfers
  - Split transactions
- APB: Advanced Peripheral Bus
  - Low power
  - Simple Interface
  - Suitable for many peripherals
  - One initiator (APB Bridge)
- AXI: Advanced eXetendible Interface
  - Configurable channel-based specification

Typical On-Chip Interconnect for Smaller Embedded Devices

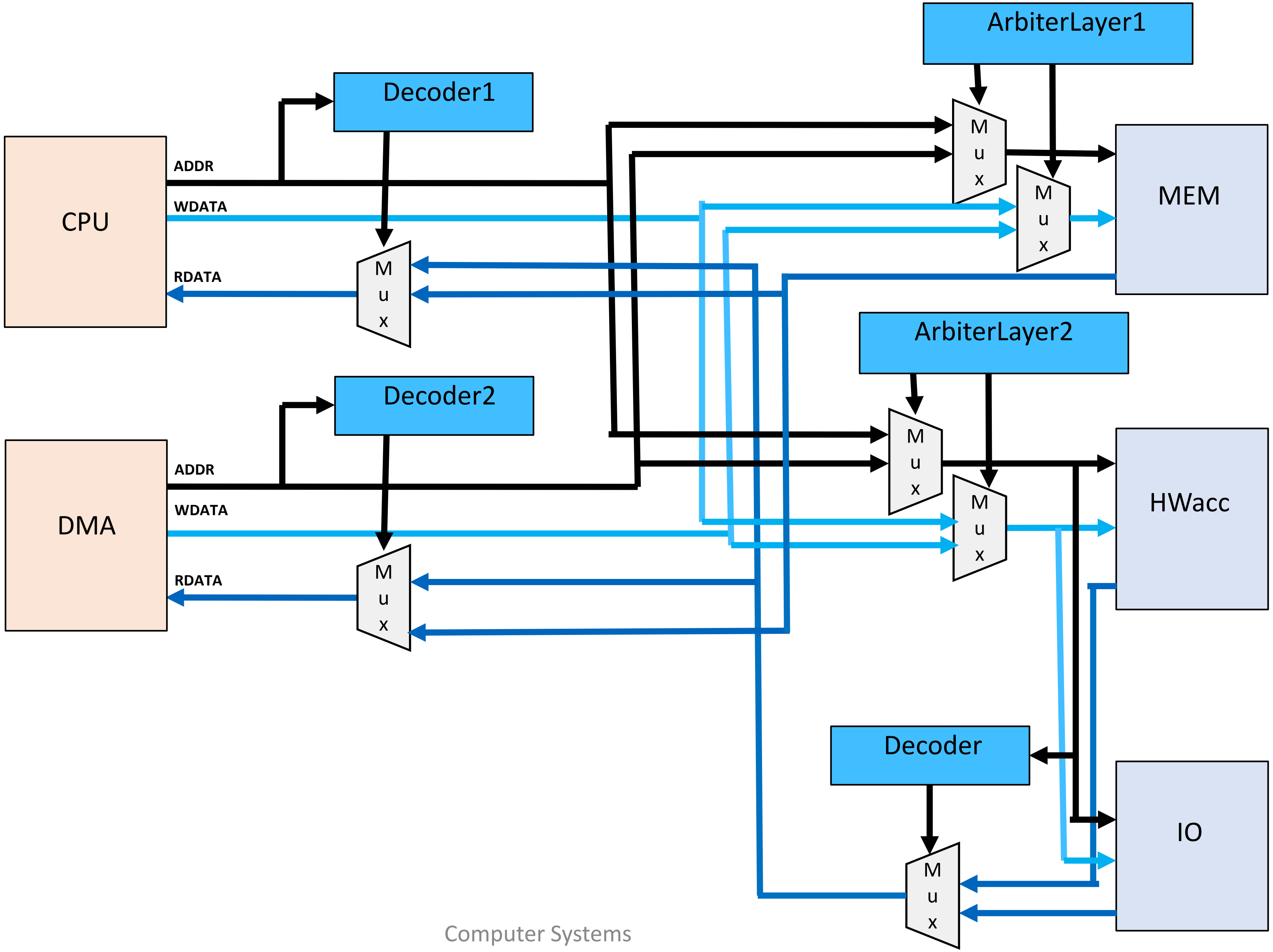
- High-performance near the processor cores, low-performance near the slow I/O devices



Example – Layered Bus

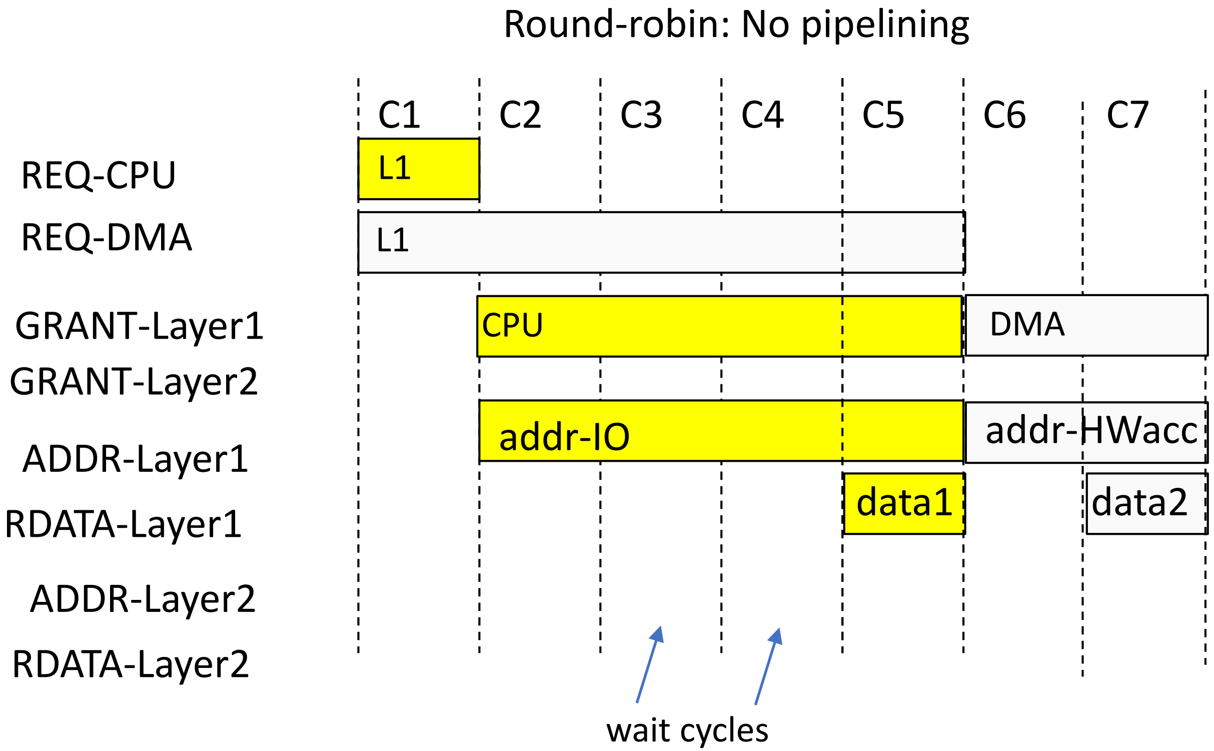
- Given is the following architecture for a shared layered bus:
  - There are two initiator components, CPU and DMA.
  - There are three target components, MEM, HWacc and IO.
    The MEM, is on layer 1, the Hwacc and IO component is on layer 2.

Example – Layered Bus



Example – Layered Bus - Access

- Assume that the CPU wants to read access the IO slave component in the bus cycle 1 and that the DMA wants to read access the HWacc in the same bus cycle 1. Draw the bus access diagram for the data and address bus of the two bus masters as well as the control request and grant signals for the two layers assuming that the bus does not support pipelining. The IO component inserts two wait cycles. The HWacc component inserts no wait cycles. The arbitration order is CPU first, then DMA. There is no pipelining.



Network-on-Chip (NoC)

Sources

- **Principles and Practices of Interconnection Networks**
  Authors: William James Dally, Brian Patrick Towles
  ISBN: 978-0-08-049780-8

- Slides inspired by the „On-Chip Networks I/II“ (L-15/L-16) lectures of Ryan Lee and Tushar Krishna: http://csg.csail.mit.edu/6.5900/lecnotes.html

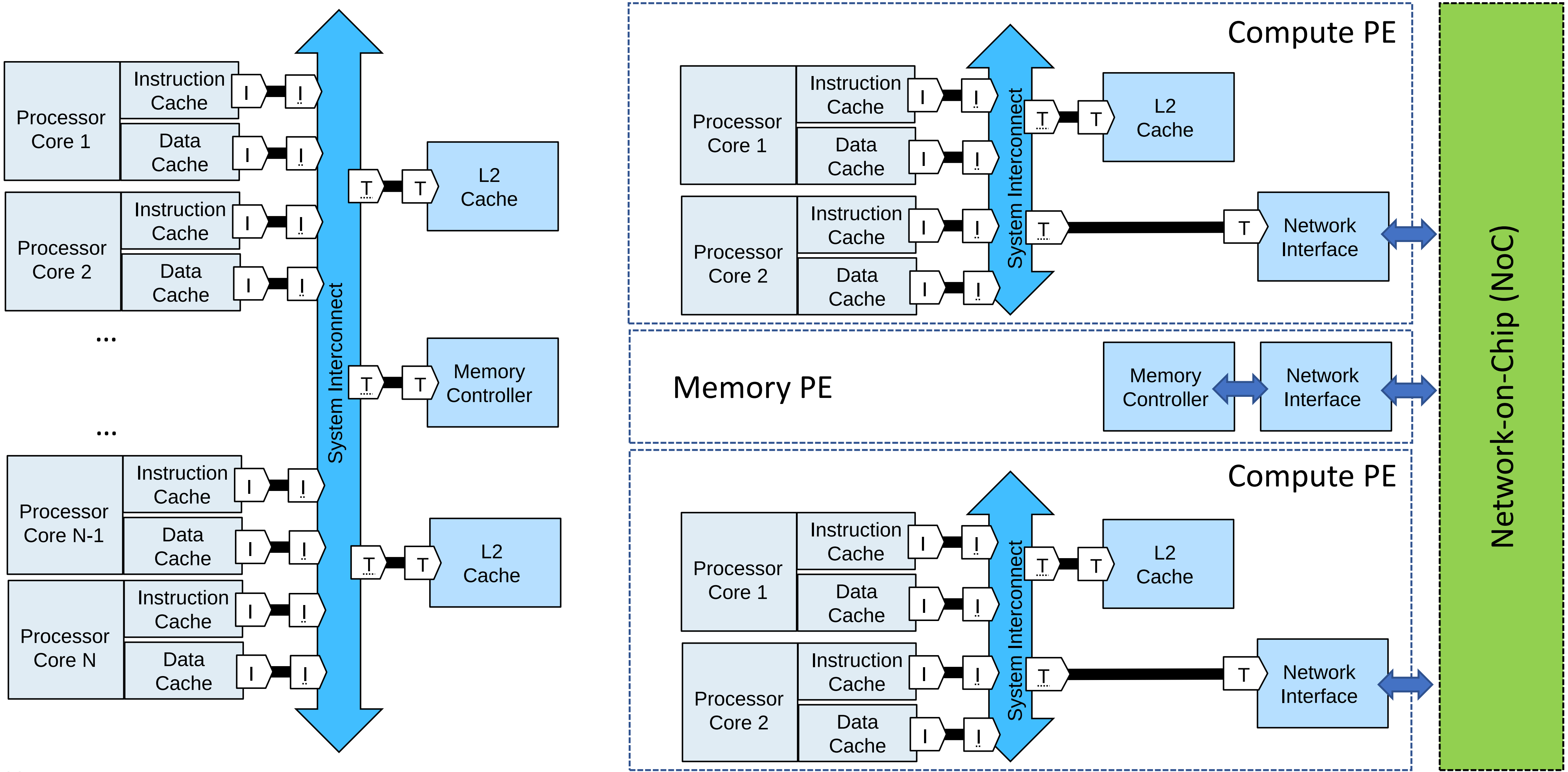
Motivation

- Need for scalability and reduced cost
  - Avoid long interconnects/delays caused by increased system complexity
  - Reduce wiring overhead caused by increasing number of system components
- Performance demands
  - Goal: high bandwidth and low latency
  - Concurrent communication required due to increased traffic

- Solution: Network-on-Chip (NoC)
  - Move from bus to network (small-scale networks on chip-/system-level)
    - Larger-scale networks in later lectures
  - Broadcast can be avoided, but still possible via multiple messages (when required)
  - Serialization achievable, e.g., by forcing the same path or via sequence numbers

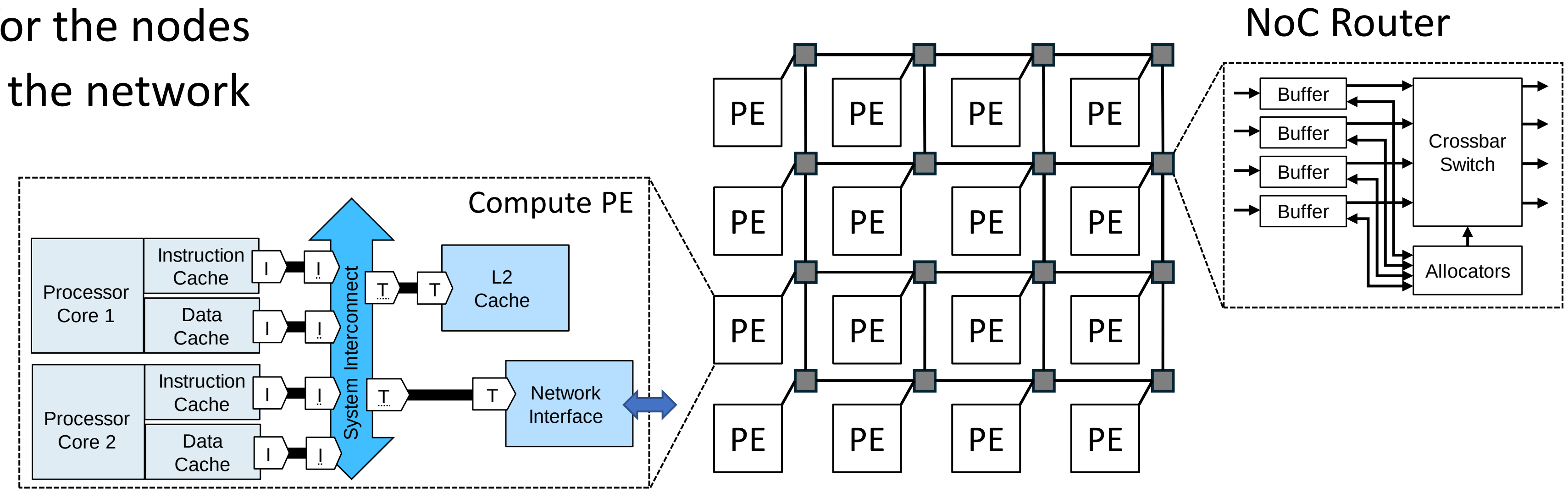
Motivation: Scalability

- Scalability: How to connect hundreds of processor cores / memory interfaces?



Network-on-Chip Basics

- Objective: Connect nodes with each other via routers and wires, so that messages can be sent from source to destination
- Building blocks:
  - Node: any component, e.g., processor, memory, or a combination of them
  - Network interface: module connecting a node to the network
  - Router: forwards data from inputs to outputs (network interfaces or other routers)
  - Link: physical set of wires, e.g., connecting two routers
  - Channel: logical connection between routers
  - Message: unit of transfer for the nodes
  - Packet: unit of transfer for the network



Design

- Topology: What is the connection pattern of the nodes?
- Routing: Which path should a message take?
- Flow control: Which network resources are granted to a message over time?

- Traffic analogy
  - Topology: defines roadmap, i.e., streets and intersections
  - Routing: steering of the car, i.e., where to turn at each intersection
  - Flow control: traffic light control, i.e., when a car can advance over the next part of the road

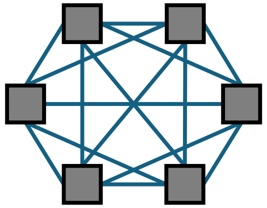
Topology

- Topology: arrangement of nodes and channels
  - Determines e.g., number of hops, number of alternative paths, cost

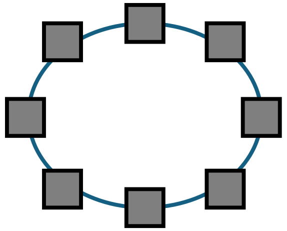
- Properties for comparison
  - Degree: number of links at each node
  - Distance: number of links in the shortest route
  - Diameter: maximum distance between any two nodes
  - Bisection bandwidth: available bandwidth from one partition to the other, when cutting the network into two equal parts (minimum for multiple possible cuts)

Topology

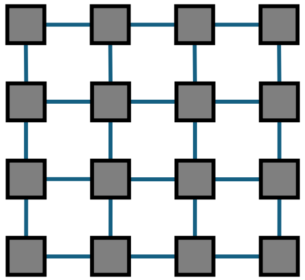
- ***Direct networks:*** each terminal node is associated with a router; routers are sources/sinks *and* switches for traffic from other nodes



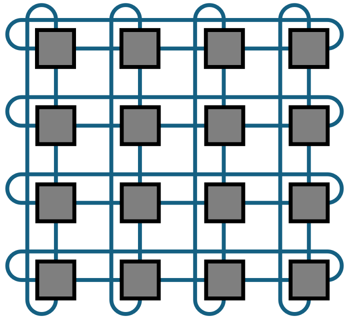
Fully Connected



Ring

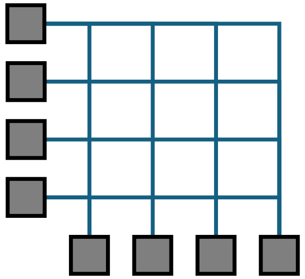


Mesh

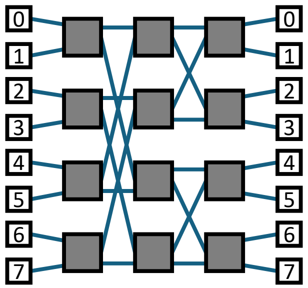


Torus

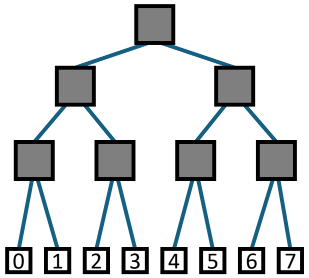
- ***Indirect networks:*** terminal nodes are connected via intermediate stages of switch nodes; terminal nodes are sources/sinks, intermediate nodes only switch traffic



Crossbar



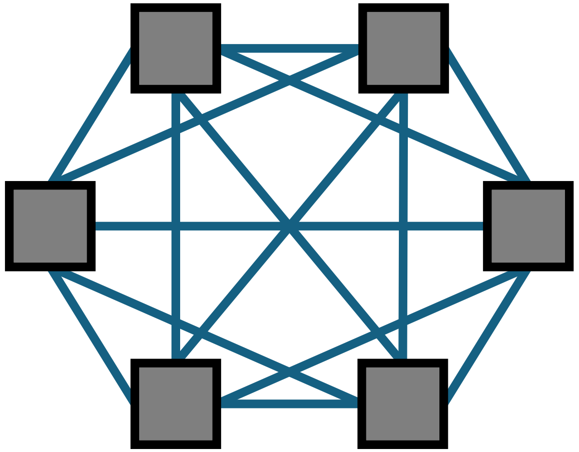
Butterfly



Tree

Fully Connected Networks

- Every node connected to every other node with a direct link
- *N* nodes, *N·(N-1)/2* links
- Degree: *N-1*
- Diameter: *1*
- Bisection width: $\lfloor N/2 \rfloor \cdot \lceil N/2 \rceil$



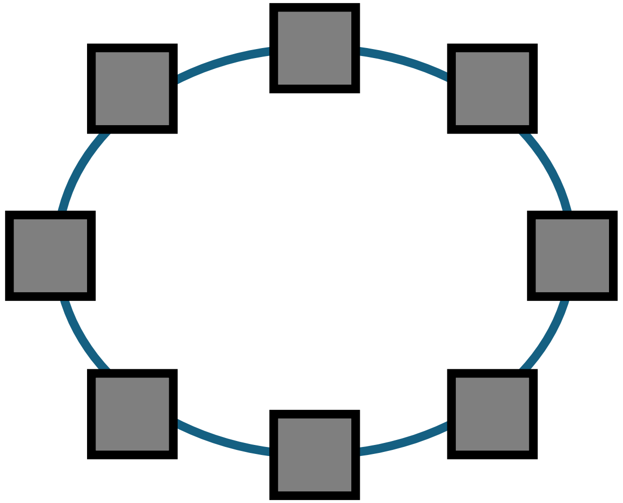
Fully Connected

- Pros: high fault tolerance, low contention, low latency
- Cons: high costs for large N, limited scalability

Ring (*k*-ary 1-cube)

- Each node connected to two other nodes
- *N* nodes, *N* links
- Degree: *2*
- Diameter: $\lfloor N/2 \rfloor$
- Bisection width: *2*

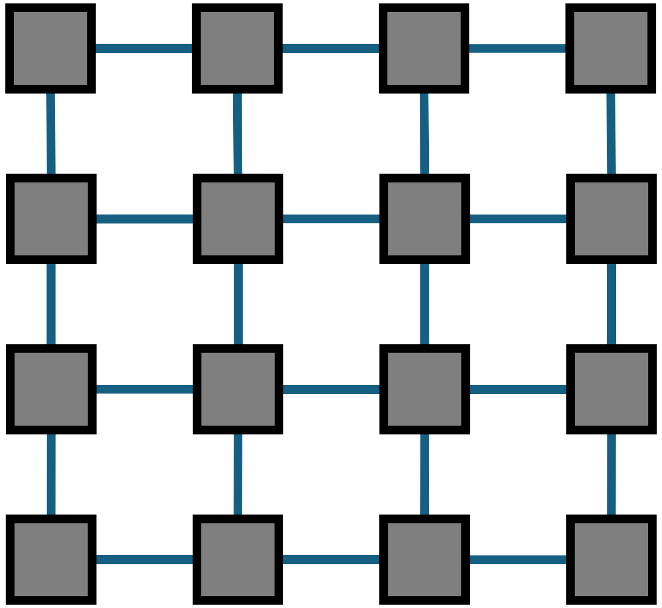
- Pros: simple, low link costs
- Cons: high latency for large N, limited path diversity



Ring

Mesh

- *k*-ary *n*-cube: *N=k*^{n} nodes in a regular *n*-dimensional grid
  - *k* nodes in each dimension
  - Links between nearest neighbors
- For n=2 (i.e., $k \times k$ grids)
  - *N=k*^{2} nodes, $2k \cdot (k-1)$ links
  - Degree: *4*
  - Diameter: *2k-1*
  - Bisection width: *k*

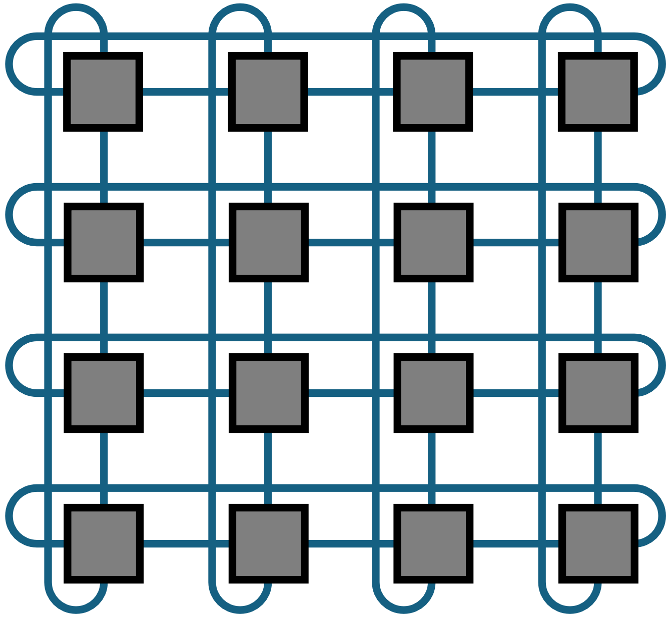


Mesh
(here: 4-ary 2-cube)

- Pros: path diversity, regular and equal-length links
- Cons: large diameter, asymmetric (higher demand for center links)

Torus

- *k*-ary *n*-cube: $N=k^n$ nodes in a regular *n*-dimensional grid
  - *k* nodes in each dimension
  - Links between nearest neighbors, adds wrap-around links at the edges compared to mesh
- For n=2 (i.e., $k \times k$ grids)
  - $N=k^2$ nodes, *2N* links
  - Degree: *4*
  - Diameter: *k*
  - Bisection width: *2k*



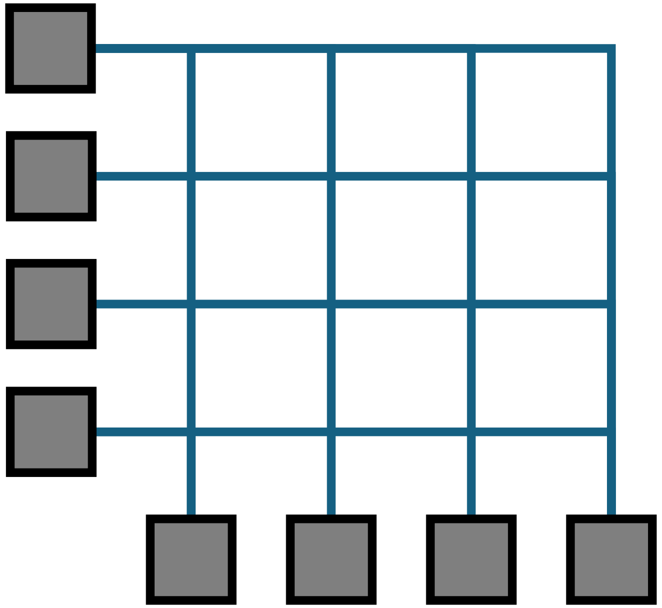
Torus

- Pros: avoids asymmetry and improves path diversity compared to mesh
- Cons: unequal link lengths and higher cost compared to mesh

Crossbar

- Connects n inputs to m outputs via $n \times m$ switches
- Switches enable concurrent communication between disjoint input/output pairs without blocking
- $N = n \cdot m$ nodes, $n \cdot m$ links
- Diameter: *1*

- Pros: non-blocking, latency (for small n, m)
- Cons: high cost, limited scalability

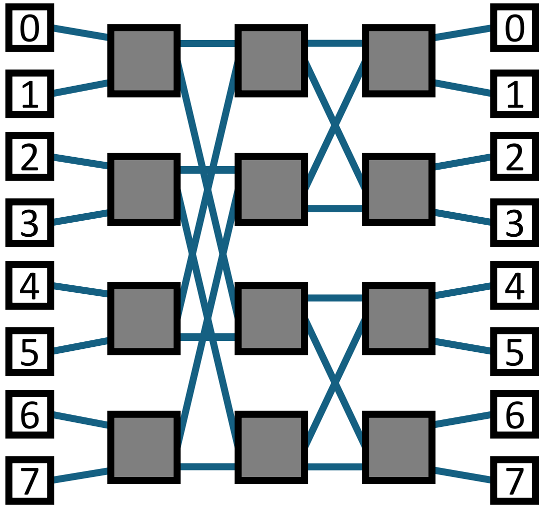


Crossbar

Butterfly

- *k*-ary *n*-flies: k^n nodes connected via *n* stages of k^{n-1} intermediate $k \times k$ switches
  - k: switch degree
  - n: number of stages of switches

- Pros: lower cost compared to crossbar
- Cons: blocking, lack of path diversity, locality not exploitable

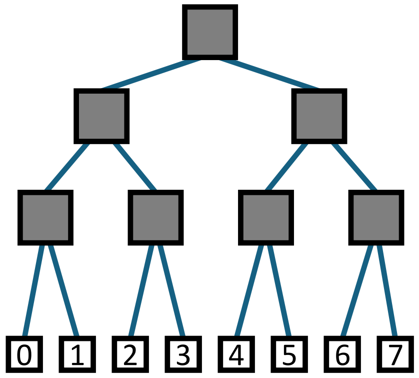


Butterfly
(2-ary 3-fly)

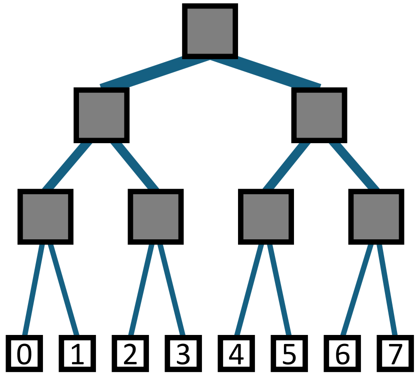
Trees

- *k*-ary tree with *N* nodes and $log_k N$ stages
- Nodes are the leaves of the tree, switches at intermediate stages
- Messages are sent up to common ancestor, then sent down to destination

- Pro: simple, cheap
- Cons: Bottleneck towards root
  - Alternative: Fat tree, where links between switches closer to the root are increased



Tree

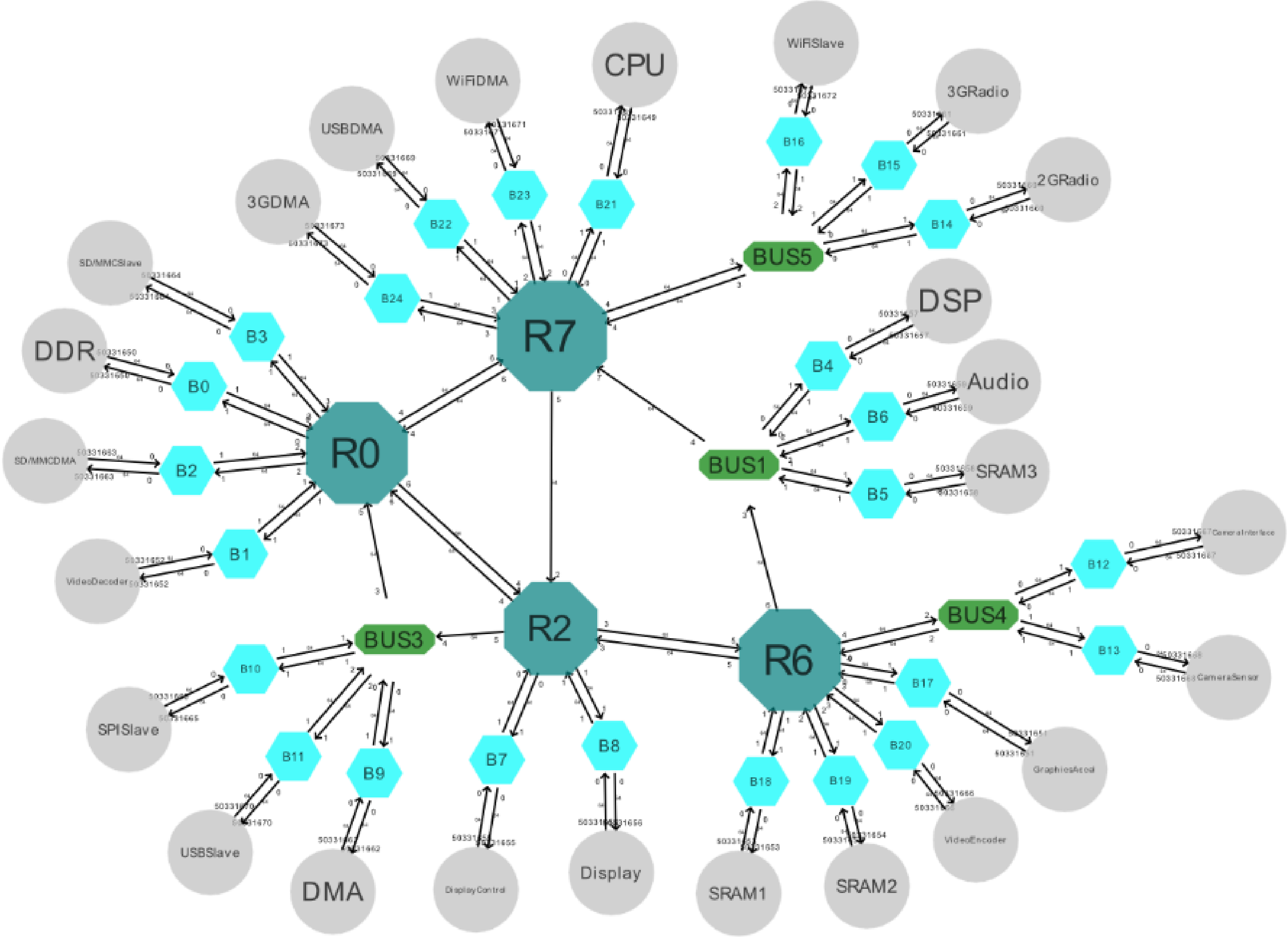


Fat tree

Application-Specific Network-on-Chip Architectures

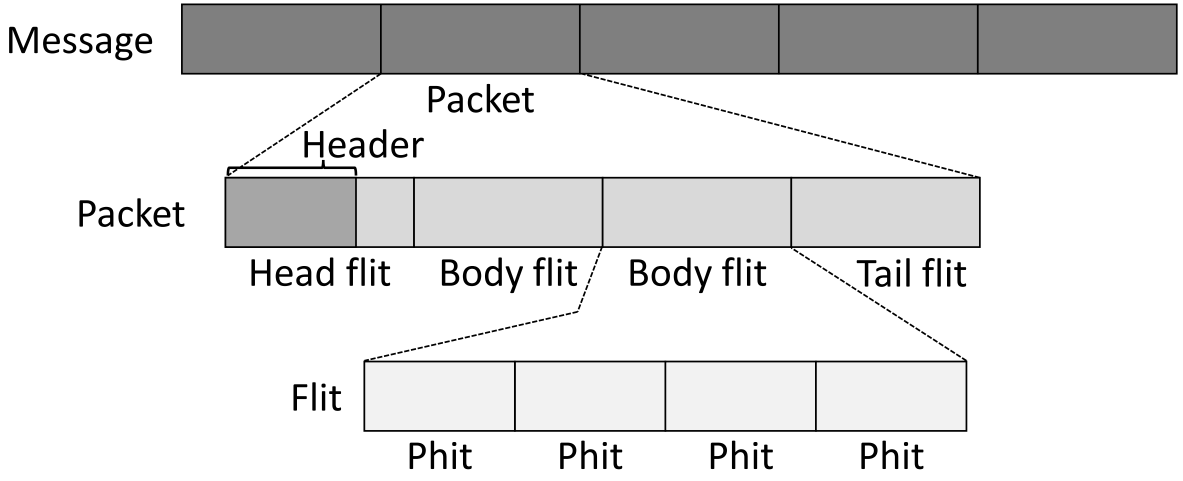
- Custom tailored NoC topology for chips with very unbalanced traffic demand for different PEs

- Example: NoC for a 3G Modem Chip (2014)



Messages

- Message: logically continuous group of bits, may be arbitrarily long
- Packet: basic unit of routing and sequencing, restricted maximum length
  - Consists of header + segment of a message
- Flit (flow control digit): basic unit of bandwidth and storage allocation
  - Contain no separate routing/sequencing information and therefore follow the same path in-order
  - Subdivision allows for low overhead (large packets) and fine-grained resource utilization (small flits)
- Phit (physical transfer digit): information transferred over a channel in a single clock cycle



Flow Control vs. Routing

- Flow control: Allocates resources (channels, control state, buffers) to packets
  - Alternative view: resolve contention during packet transmission
  - Contention: What happens if two packets want to use the same channel at the same time?
- Routing: Selects the path a packet takes from source to destination
  - Determines how well the potential of the given topology is exploited
  - Should balance load across network channels

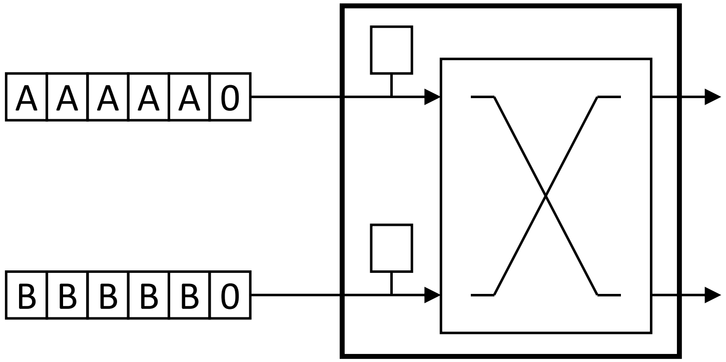
Flow Control

- Bufferless
  - Dropping
  - Misrouting
  - Circuit switching
- Buffered
  - Store-and-forward
  - Cut-through
  - Wormhole
  - Virtual channel

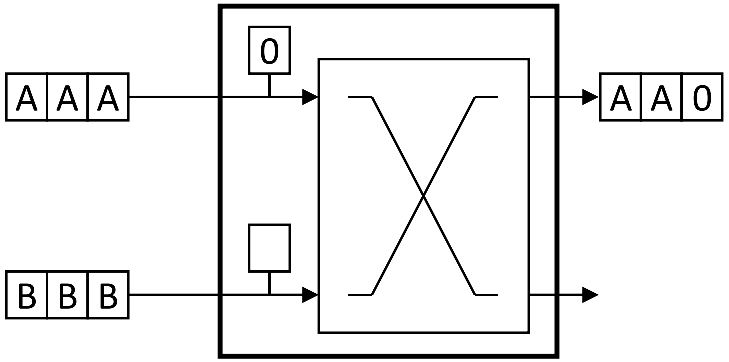
Bufferless Flow Control: Dropping

- Competing packets: No buffers available, therefore drop “losing” packets, “winning” packet is allowed to proceed
- Example:

Two packets A and B arriving, both requesting channel 0



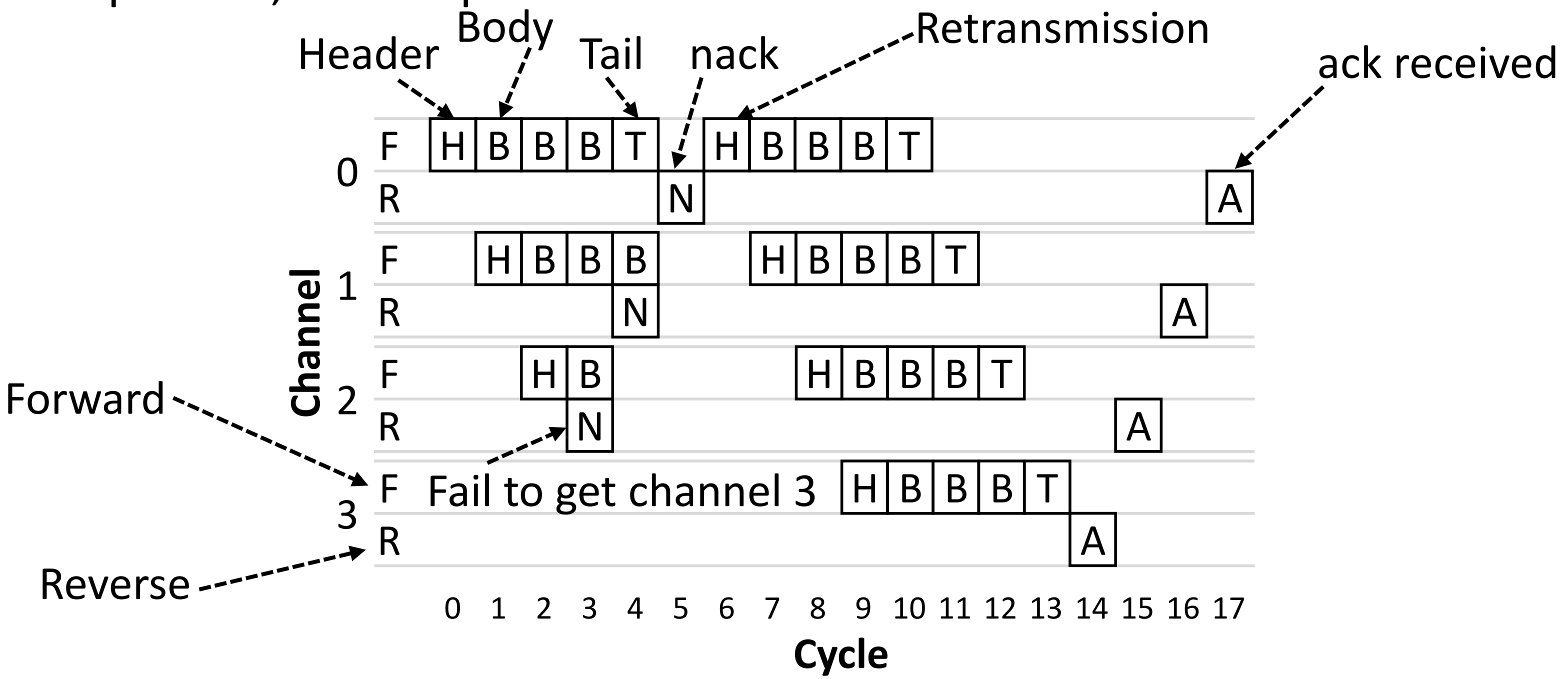
Packet A “wins”, B is dropped and must be retransmitted from source



  - Complete effort already invested in packet B is lost
  - Source needs to be informed to about successful transmission or need for retransmission

Bufferless Flow Control: Dropping

- Time-space diagram with negative acknowledgements (nacks)
  - Example: five-flit packets, four-hop route



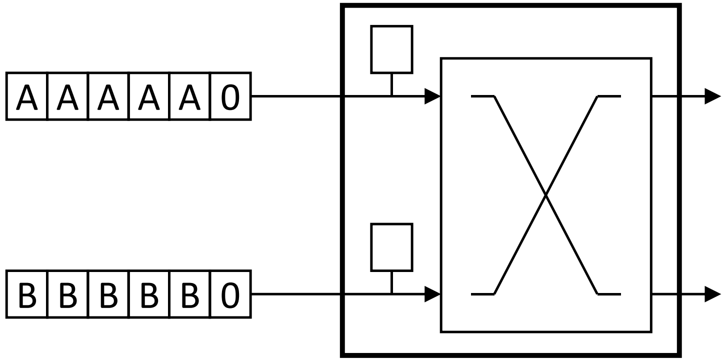
- Alternative: no nacks, resend packet if ack is not received before a timeout

- Dropping: simple, wastes resources

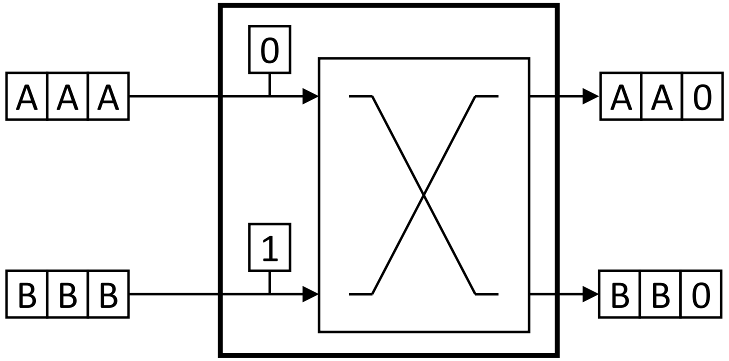
Bufferless Flow Control: Misrouting

- Competing packets: No buffers available, therefore misroute “losing” packets, “winning” packet gets the requested channel
- Example:

Two packets A and B arriving, both requesting channel 0



Packet A “wins”, B is misrouted to channel 1

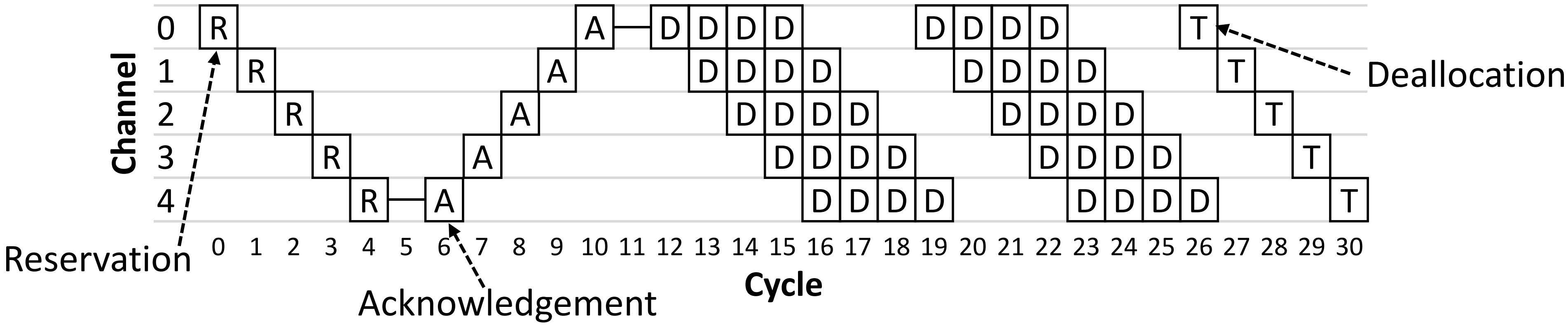


  - Requires sufficient path diversity
  - Routing needs to ensure that packet reaches its destination despite misrouting

- Misrouting: no packet dropping, packets sent in wrong direction, livelock possible (need to guarantee forward progress)

Bufferless Flow Control: Circuit Switching

- First allocate channels to build a circuit from source to destination, then send packets along the circuit, deallocate circuit after packets are sent
- Example: four-flit packets, five-hop route
  - 1. Send request (R) to destination allocating channels along the way
  - 2. Destination returns acknowledgement (A) to source
  - 3. Data flits (D) are sent
  - 4. Tail flit (T) deallocates the channel



Channel	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30
0	R										A		D	D	D	D				D	D	D	D				T				
1		R								A				D	D	D	D				D	D	D	D				T			
2			R						A						D	D	D	D				D	D	D	D				T		
3				R				A								D	D	D	D				D	D	D	D				T	
4					R		A										D	D	D	D				D	D	D	D				T

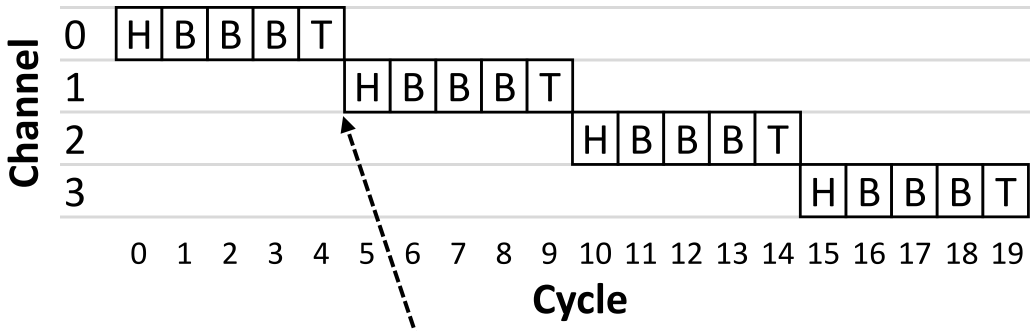
- Circuit switching: simple, high latency, high overhead for circuits with short duration

Buffered Flow Control

- Buffers allow to store data while waiting for the following channel
  - Without buffers data arriving at cycle *i* had to be transmitted at cycle *i+1* (or dropped)
- Flow control now needs to allocate channels *and* buffers
  - Allocation at packet or flit granularity
  - Packet granularity: store-and-forward, cut-through
  - Flit granularity: wormhole

Buffered Flow Control: Store-and-forward (Packet-based)

- Each node waits until packet is received completely before transmission to the next node
- Need to allocate channel and sufficient buffer space for the packet in the next node
- Example: five-flit packet, four-hop route without contention

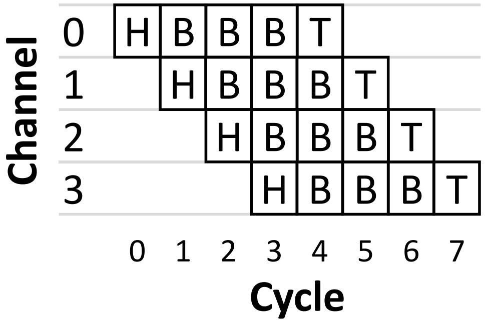


Could also be transmitted later if channel/buffer space is not available

- Store-and-forward: channels not held idle, only small buffers required, high latency due to serialization

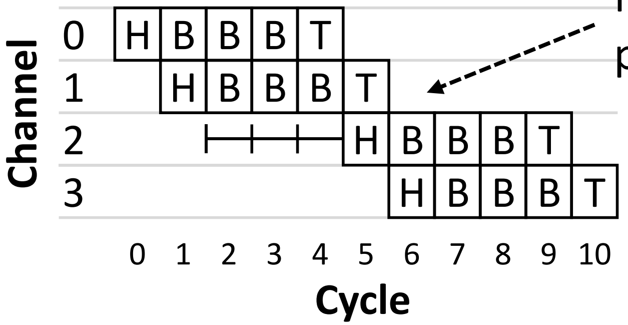
Buffered Flow Control: Cut-through (Packet-based)

- Flits are forwarded as soon as they are received *and* the following channel and buffer space is acquired (allocation still at packet granularity)
- Avoids waiting for receiving the complete packet before transmission
- Example: five-flit packet, four-hop route without/with contention



Channel	0	1	2	3	4	5	6	7
0	H	B	B	B	T			
1		H	B	B	B	T		
2			H	B	B	B	T	
3				H	B	B	B	T

No contention



Channel	0	1	2	3	4	5	6	7	8	9	10
0	H	B	B	B	T						
1		H	B	B	B	T					
2			├	┼	┼	H	B	B	B	T	
3							H	B	B	B	T

No breaks *within* packet transmission

Three-cycle contention before channel 2

- Cut-through: high channel utilization, low latency, inefficient use of buffer storage and long contention latency due to packet-based allocation

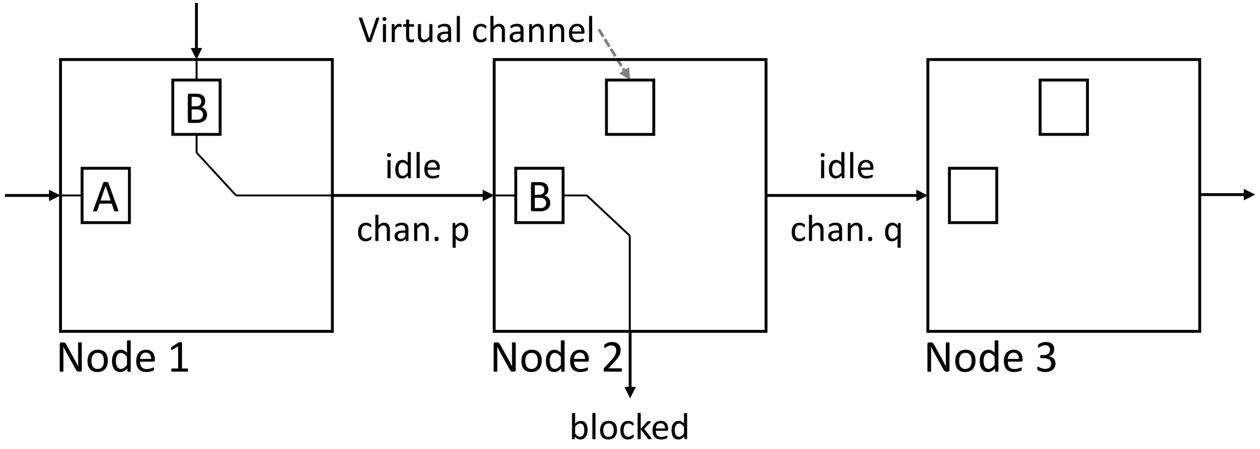
Buffered Flow Control: Wormhole (Flit-based)

- Similar to cut-through, but allocates channels and buffers to flits instead of packets
  - Head flit requests channel state (virt. channel) for the *packet*, buffer for one flit and channel for one flit
  - Body flits use virtual channel to follow head flit, request buffer for one flit and channel for one flit
  - Tail flit treated like body flit, but additionally releases virtual channel
- Blocking might occur as the single virtual channel belongs to a packet, while buffers are allocated to flits
  - Channel set to idle if buffer cannot be acquired (it cannot be used by other packet)
- Wormhole: Saves buffer space, may block a channel mid-packet

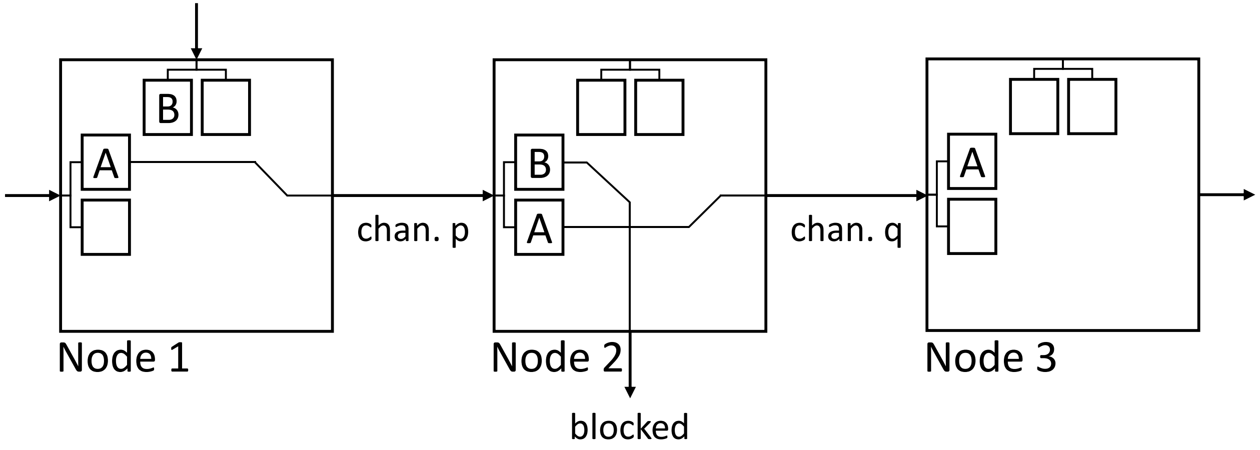
- Improvement: virtual-channel flow control
  - Associate multiple virtual channels (channel state and flit buffers) with single physical channel
  - Other packets can use channel when one packet is blocked
  - Competition for transmitting flits over single physical channel
  - Reduces blocking, more complex routers

Buffered Flow Control: Wormhole vs. Virtual-channel

- Wormhole flow control: When B blocks, channel p and q are idle

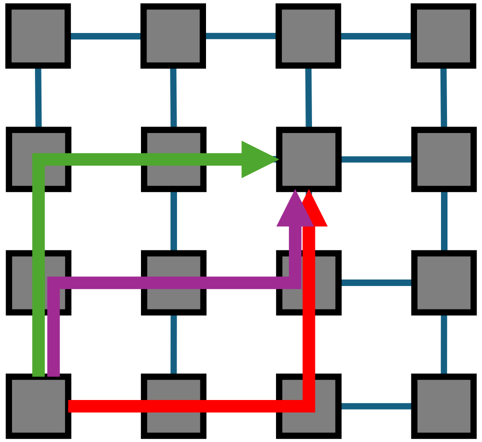


- Virtual-channel flow control: A can use channel p and q using a second virtual channel



Routing

- Selects the path a packet takes from source to destination in a given topology
- Determines how well the potential of the given topology is exploited



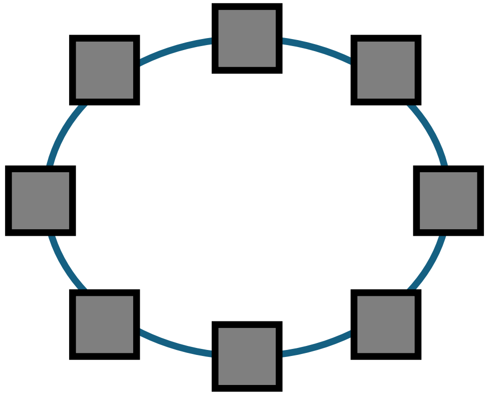
- Balance load across the network channels to avoid hotspots and contention
  - Difficult, particularly with non-uniform traffic patterns causing load misbalances

Routing Algorithms

- Properties
    - Minimal or non-minimal
        - Minimal: select shortest paths
        - Non-minimal: not limited to shortest paths only
    - Oblivious or adaptive
        - Oblivious: select route without considering information about current network state
            - Deterministic: Subset of oblivious; always select same path between source and destination
        - Adaptive: select route based on current network state
- Design aspects
    - Table-based or algorithmic
        - Table-based: Table lookup of the entire route (source-table routing) or at each node along the route (node-table routing)
        - Algorithmic: Compute route using an algorithm usually implemented via combinational logic
    - Deadlocks
        - Situations where packets cannot make progress as they are waiting on one another to release resources

Routing Example

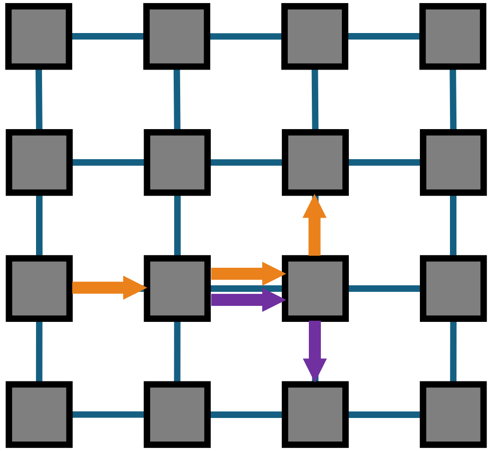
- Routing decision in ring network: clockwise or counter-clockwise?



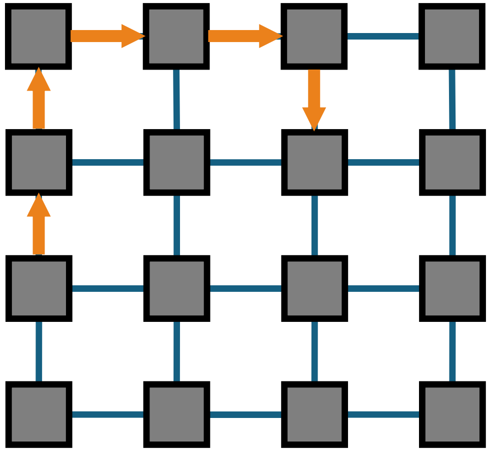
- Potential routing algorithms
  - Greedy (deterministic, minimal): always pick the shortest direction
  - Uniform random (oblivious, non-minimal): randomly pick a direction with equal probability
  - Weighted random (oblivious, non-minimal): randomly pick a direction with a higher weight for shorter direction
  - Adaptive (adaptive, non-minimal): pick direction based on load of the local channels

Dimension-order Routing

- First move towards x-dimension, then move towards y-dimension (XY)
  - To increase the clarity, we will focus on 2D meshes in the following
- Example: 2D Mesh



Dimension-order routing:
Deterministic and minimal

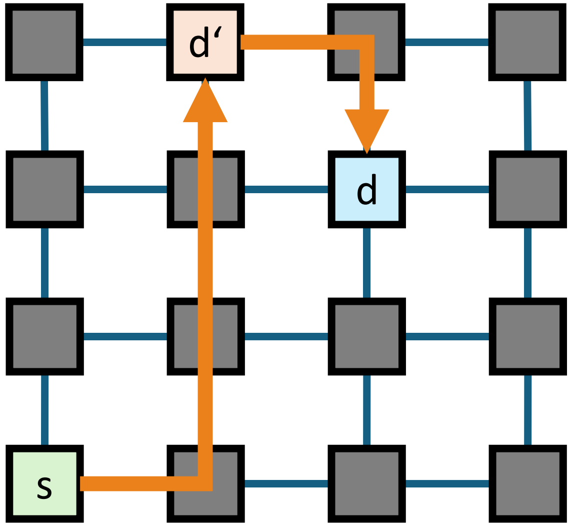


Alternate route:
non-minimal

- Dimension-order routing: simple, minimal, can cause load imbalance, doesn‘t exploit path diversity

Valiant‘s Algorithm

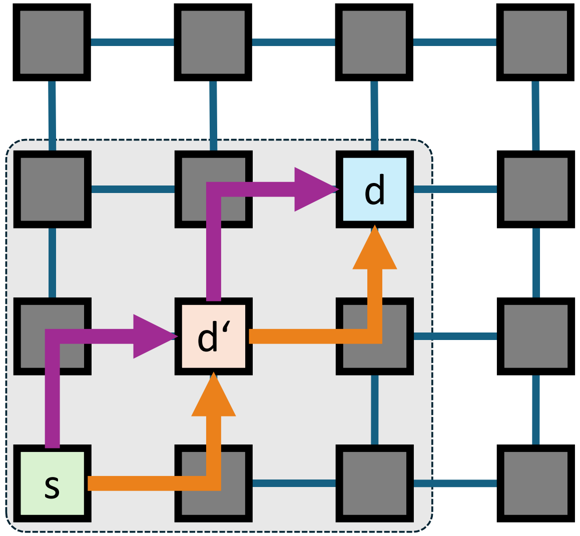
- Packet from source *s* to destination *d* is routed via an intermediate node *d‘*
  - Randomly select intermediate node *d‘*
  - Phase I: Route packet from *s* to *d‘*
  - Phase II: Route packet from *d‘* to *d*
  - Use arbitrary routing algorithm for Phase I+II, e.g., dimension order routing for tori and meshes
- Can use arbitrary routing algorithm for the two phases
  - For tori and meshes: Dimension-order routing as appropriate choice



- Valiant‘s Algorithm: Randomizes traffic, balances network load, non-minimal, doesn‘t exploit locality

Valiant‘s Algorithm

- Minimal version of Valiant‘s algorithm for k-ary n-cubes:
  - Restrict intermediate node: *d‘* lies in minimal quadrant between *s* and *d* (subnetwork with *s* and *d* as corner nodes)
  - Randomly selects among minimal routes
- Steps:
  - Identify quadrant
  - Select intermediate node *d‘* from quadrant
  - Route from *s* to *d‘*
  - Route from *d‘* to *d*
- With dimension-order routing (either XY or YX): Doesn‘t use all paths
  - Idea: Select randomly whether to use XY or YX (but: deadlock problem arises)

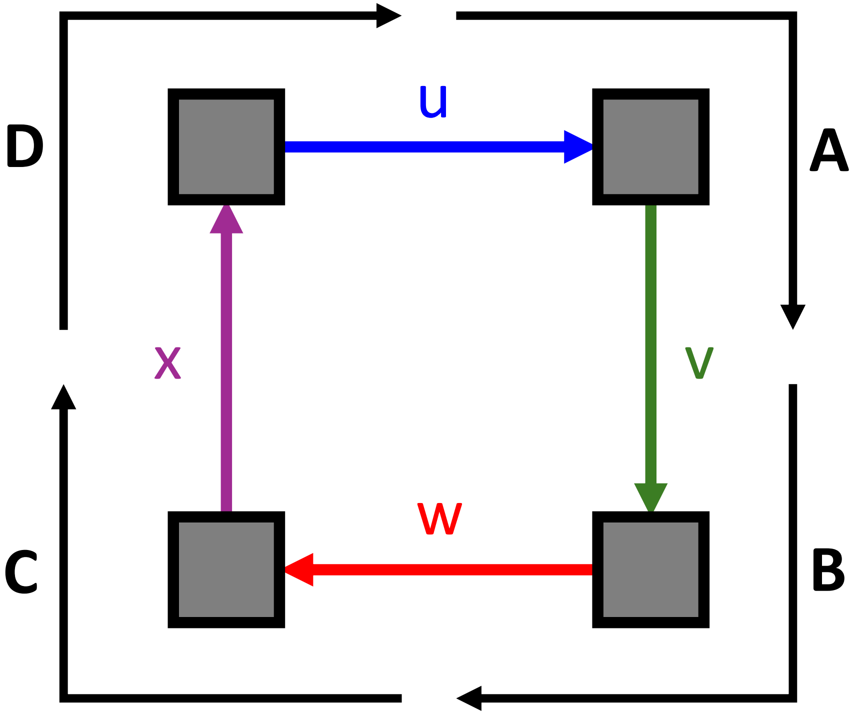


- Preserves locality, improves load balancing (compared to deterministic routing)

Deadlocks

- Deadlock: Situation where packets cannot make progress as they are waiting on each other to release resources (buffers or channels)

- Example:
  - Nodes: 0, 1, 2, 3; Channels: u, v, w, x
  - A holds *u* and waits for *v*
  - B holds *v* and waits for *w*
  - C holds *w* and waits for *x*
  - D holds *x* and waits for *u*

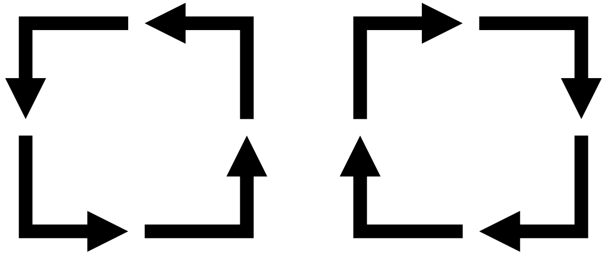


- Observation: Cycles pose a problem

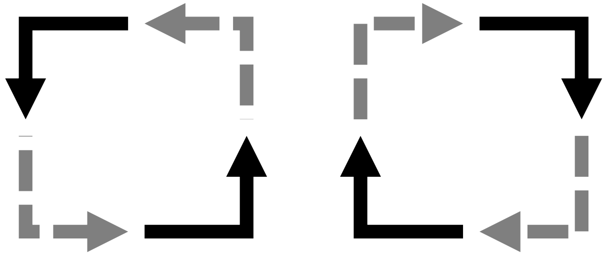
Deadlock Avoidance: Restrict Routing

- Dimension Order Routing (k-ary n-meshes)
  - E.g., first x then y (we have seen this approach already)
  - Deadlock-free, but restricts path diversity

- Turn Model: Focuses on the turns allowed and the cycles they can form
  - 2D mesh: 8 possible turns forming two abstract cycles

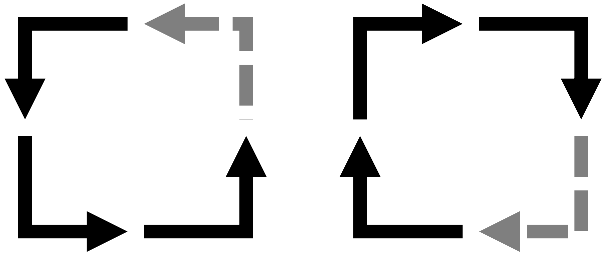


  - XY Routing removes four turns (prevents deadlocks)

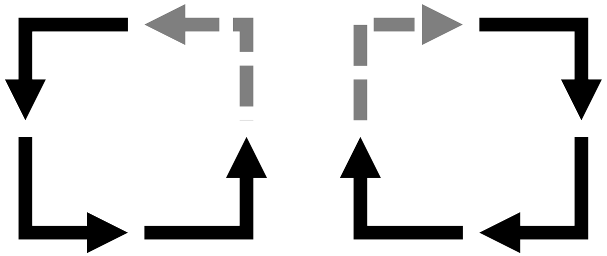


Deadlock Avoidance: Restrict Routing

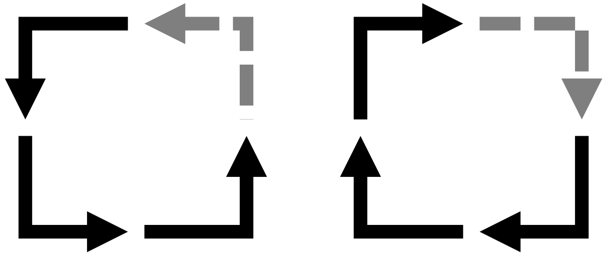
- Turn Model: Focuses on the turns allowed and the cycles they can form
  - Removing one (carefully selected) turn from each abstract cycle also prevents deadlocks



west-first: traveling west only allowed at the start

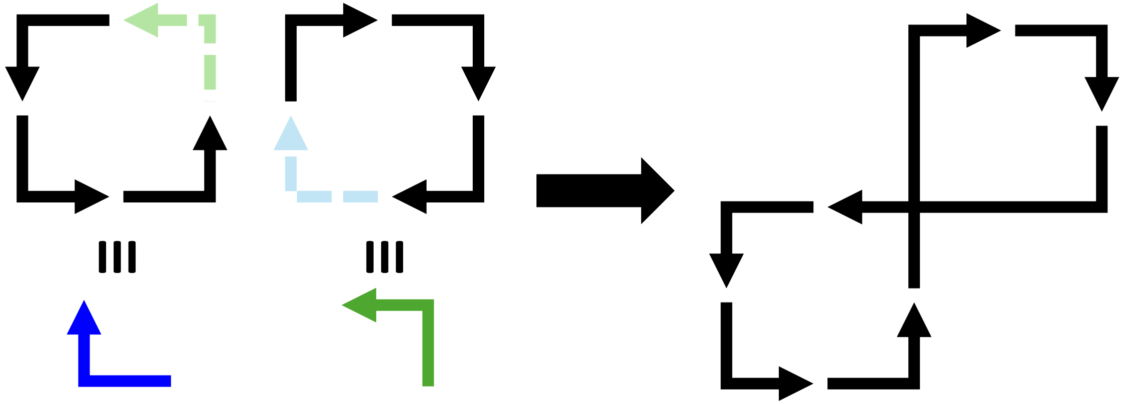


north-last: traveling north only allowed as last direction



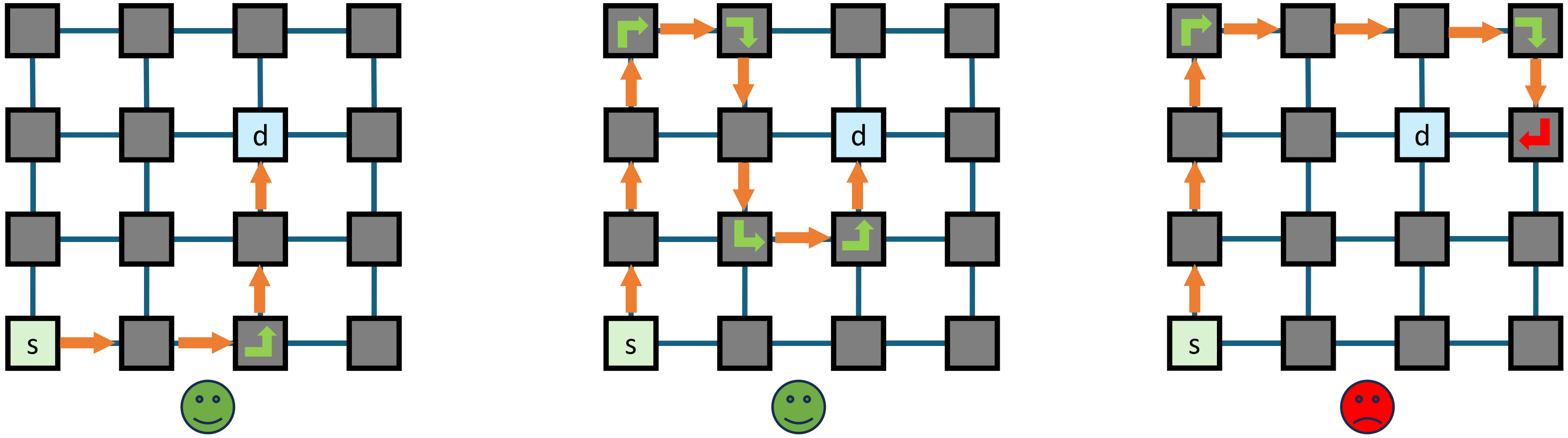
negative-first: traveling first west and south, then east and north

  - Removing any two turns does not prevent deadlocks

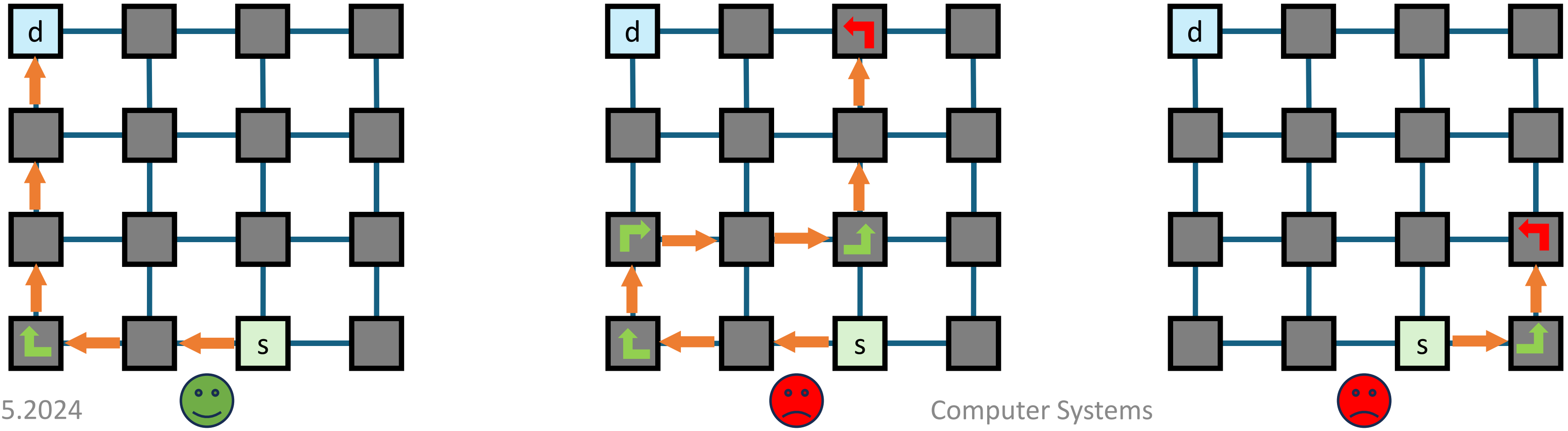


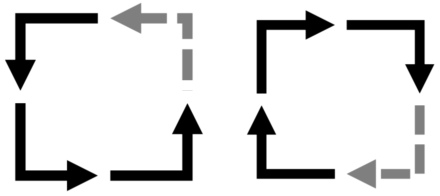
Examples: West-First

- Example 1

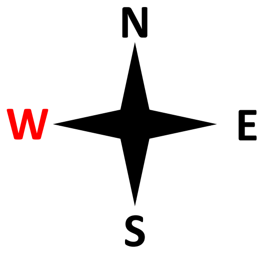


- Example 2



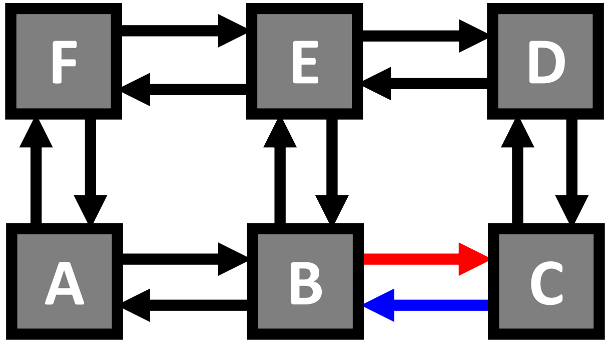


west-first: traveling west only allowed at the start

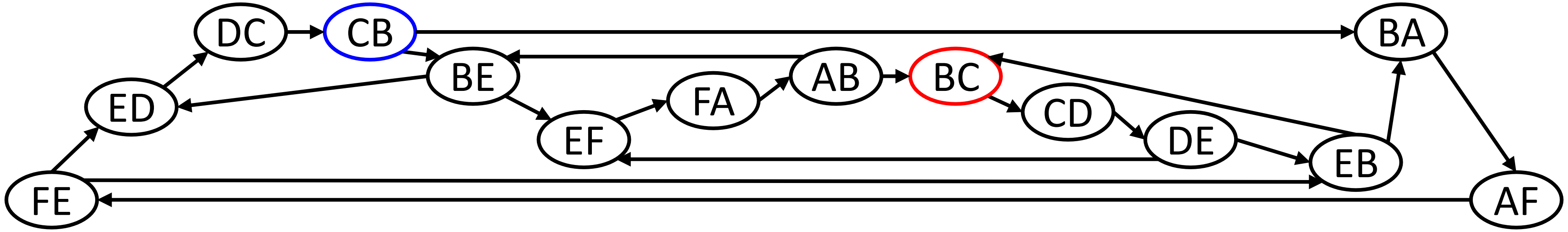


Channel Dependence Graph (CDG)

- Network topology:

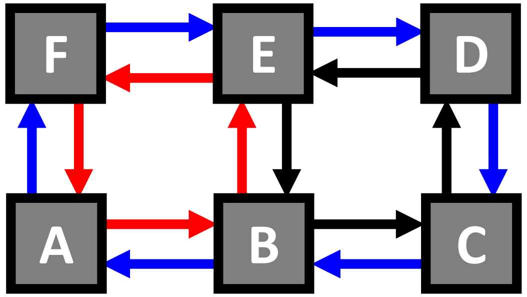


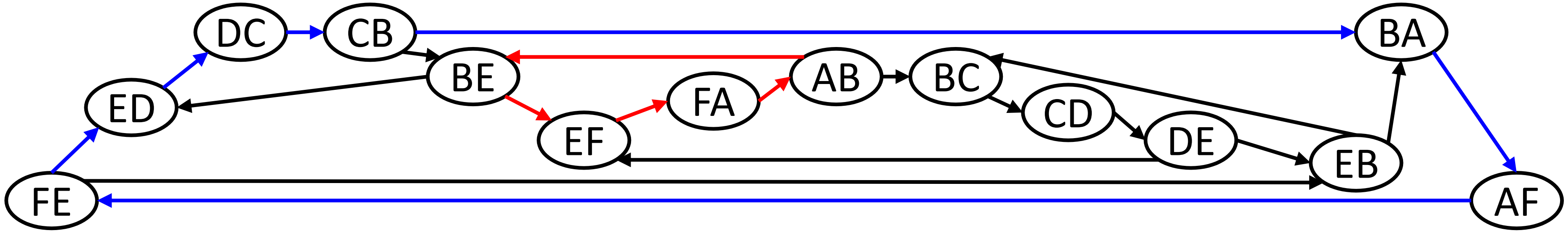
- Channel Dependence Graph:
  - One vertex for each channel
  - Edges denote dependences
    - Dependence exists if it is possible for channel *i* to wait for channel *i+1*
    - 180° turns not allowed (e.g., AB → BA)



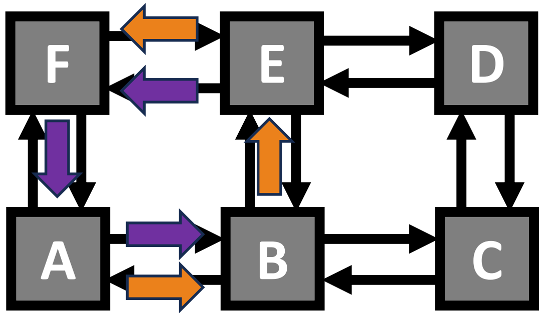
Cycles in the CDG

- Channel Dependence Graph may contain cycles





- Route through AB, BE, EF and route through EF, FA, AB → Deadlock

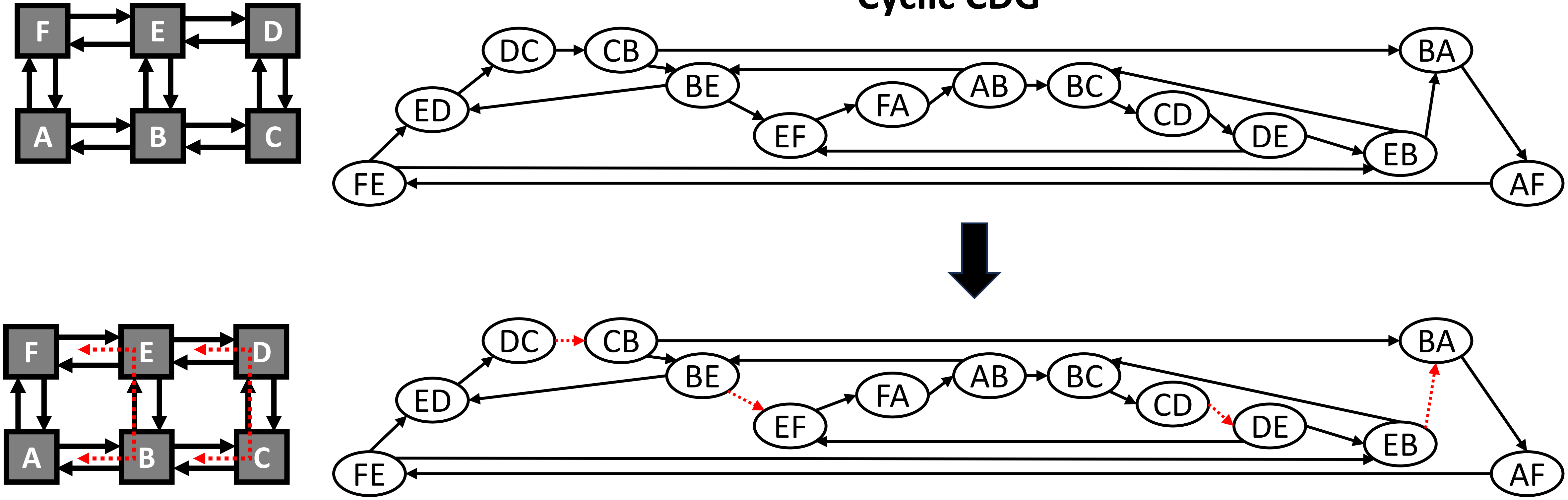


→ *Remove selected edges in the CDG*

Acyclic CDG

- Example: Remove Edges in the CDG (West-first turn model)

Cyclic CDG



Acyclic CDG

A look at real Systems-on-Chip

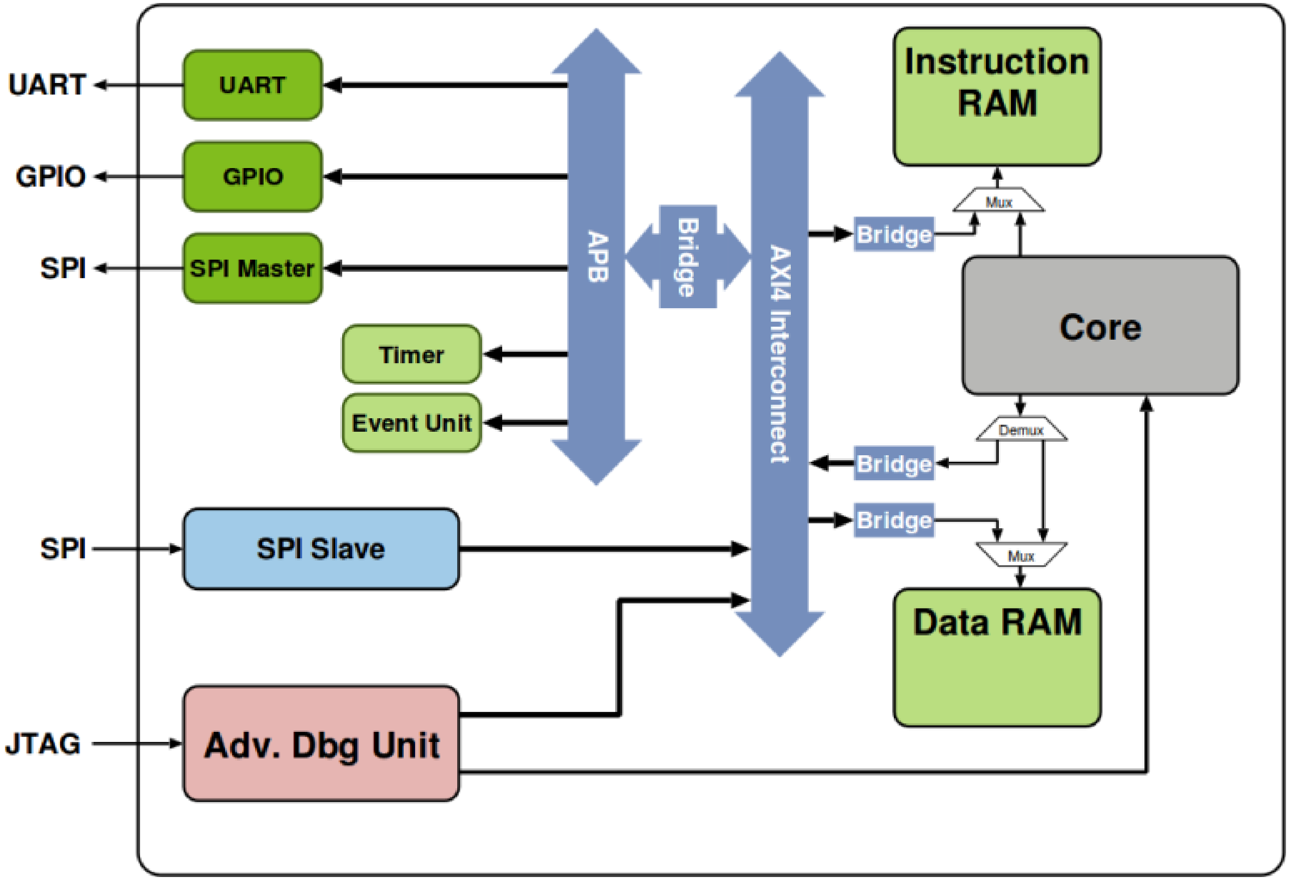
Optional, not relevant for exam

PULP 2016, PULP 2022, SpiNNaker2

Simple SoC Architecture for IoT / Wearables – Example - PULPino 2016

- SoC: System-on-chip
- PULPino Architecture 2016:
  All memories are on the same chip as the processor core

- SoC Modules:
  - Processor Core
  - Instruction memory
  - Data memory
  - Input/output devices: UARR, SPI, GPIO
  - Timer
  - Programming and Debug Devices: SPI Slave and Debug Unit
  - Connected by **on-chip interconnect:** AHB, AXI4

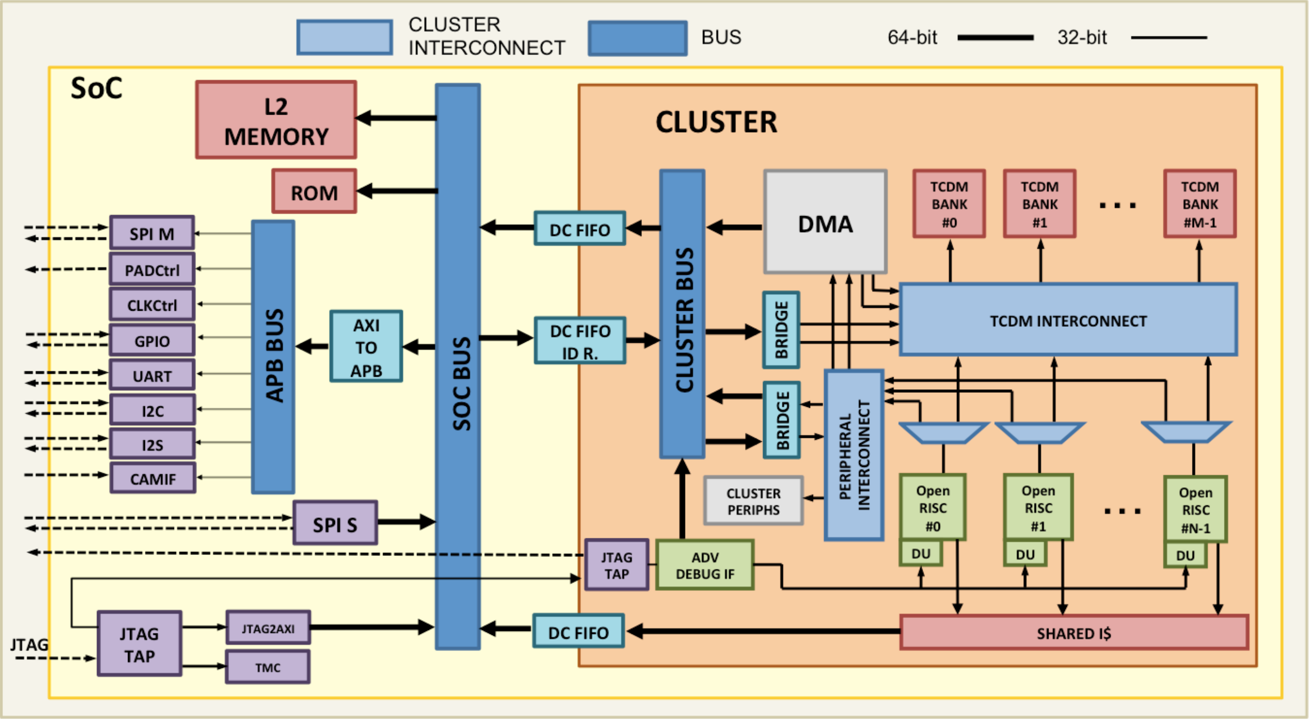


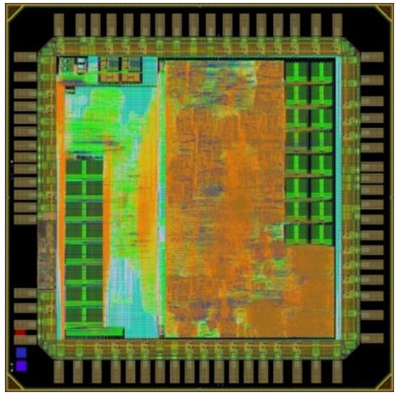
Source: CNX Software
https://www.cnx-software.com/2016/04/06/pulpino-open-source-risc-v-mcu-is-designed-for-iot-and-wearables/

Complex Multi-core SoC – Example PULP 2022

- More complex architecture
- Different **On-chip Interconnects**
- DMA: Direct Memory Access – Module to offload data movements from the CPU
- Multi-Core with shared caches

- All these modules are physically integrated in one integrated circuit (IC).

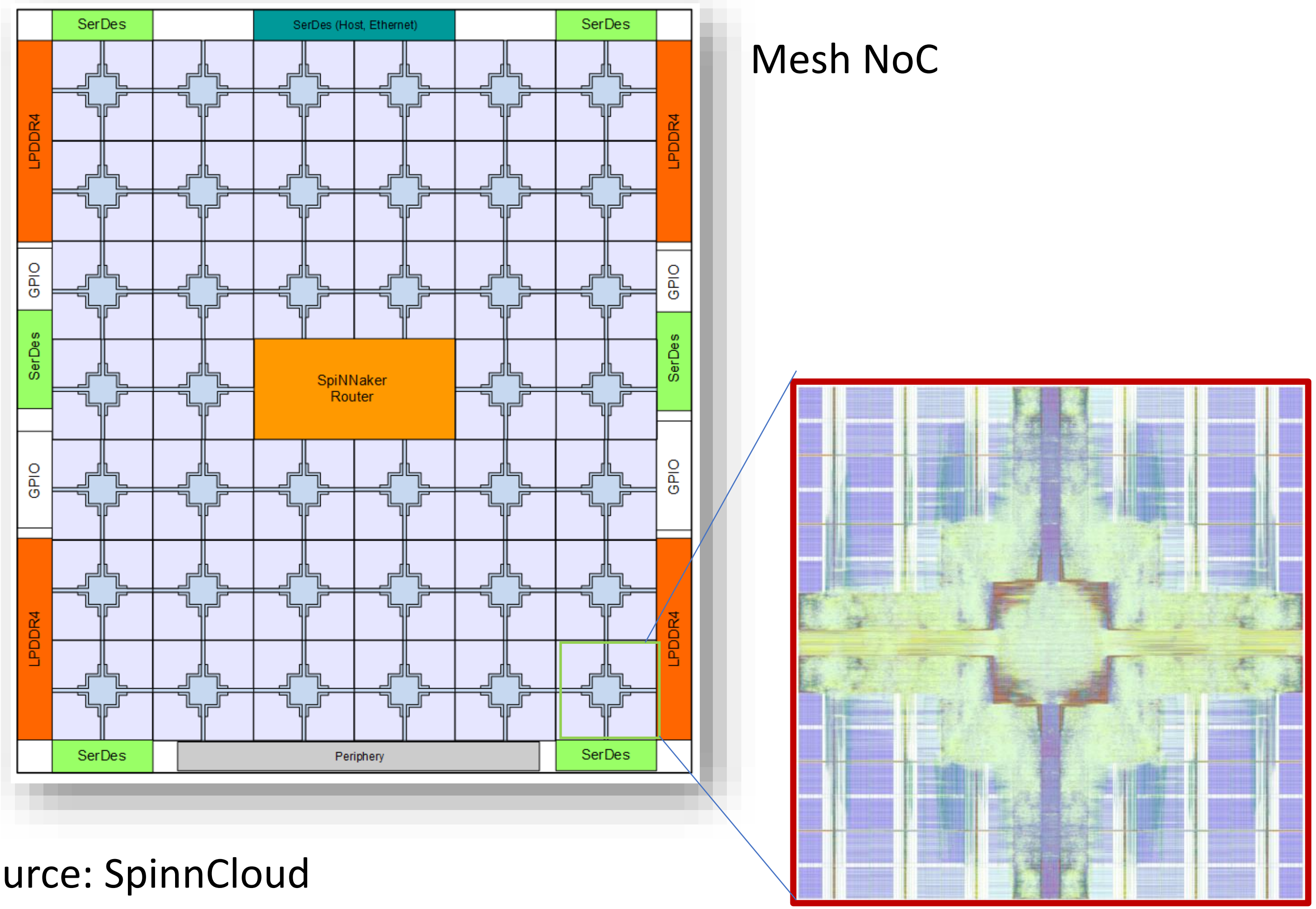




Source: https://iis-projects.ee.ethz.ch/index.php/PULP

SpiNNaker2 Chip

- Brain-inspired Chip designed for Spiking Neural Netwoks (SNNs)



Mesh NoC

Source: SpinnCloud

Summary

Conclusion

- Bus-based On-chip Interconnect
- Network on-Chip
- Next Sessions: Specialized Cores

Thank you for your attention

Computer Systems

Heterogeneous Systems-on-Chip 2 – Vector Processors

Daniel Mueller-Gritschneder

23.05.2024

Heterogene Systems-on-Chip (SoCs)

- SoCs are often multi-core systems
- General-purpose SoCs may have many replications of general-purpose processors (e.g. many ARM or standard RISC-V cores)
- To improve energy-efficiency many SoC use specialized cores (heterogeneity).

Types of Specialized Cores

- Vector Processors:
  - Introduced in the 70ties (Cray)
  - Got new attention recently especially due to machine learning workloads (x86, ARM and RISC-V Vector Instructions)
- GPUs:
  - GPUs were initially introduced for rendering graphics in real time especially for video games.
  - General Purpose (GP-GPU): Programming Language such as CUDA from NVIDIA allowed to use GPUs for other compute besides rendering (also a lot for machine learning)
- HW Accelerators:
  - Processing Cores that are specialized for a certain task (with very limited programmability)
  - Usually faster and more energy efficient than software running on programmable core
  - Different types:
    - Deep Learning: Tensor Processing Units / Neural Processing Units
    - Security: Encryption & Decryption
    - Video En/Decoders
- Application-specific Instruction Set Processors (ASIPs)
  - Between general-purpose programmable cores and accelerators
  - Some programmability but tailored towards a certain application
  - Example: Audio/Video Digital Signal Processors (DSPs)

Agenda

- Flynn’s Taxonomy
- Vector Units
- RISC-V Vector Instruction Set
- Vectorization
- Packed SIMD

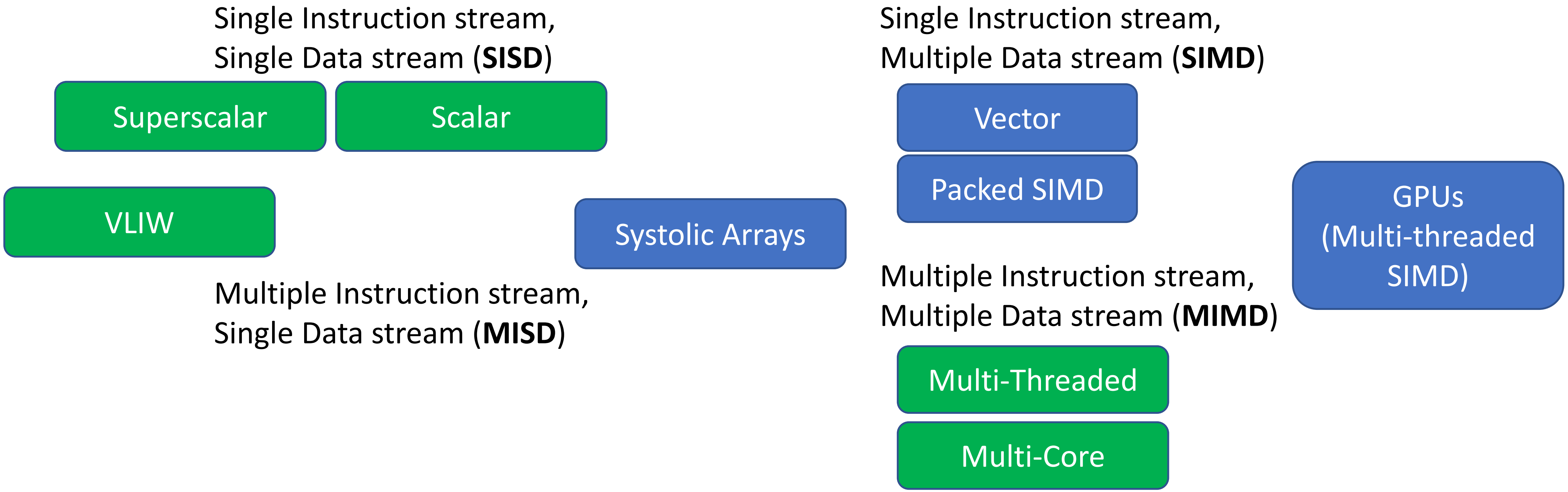
Optional, not relevant for exam

- A look at a real vector unit: ARA

Flynn’s Taxonomy

Flynn’s Taxonomy

- Classification of Computing Cores



Single Instruction stream,
Single Data stream (**SISD**)

Superscalar

Scalar

VLIW

Systolic Arrays

Multiple Instruction stream,
Single Data stream (**MISD**)

Single Instruction stream,
Multiple Data stream (**SIMD**)

Vector

Packed SIMD

GPUs
(Multi-threaded
SIMD)

Multiple Instruction stream,
Multiple Data stream (**MIMD**)

Multi-Threaded

Multi-Core

Vector Units

Vector Instruction Sets

- One instruction operates on several data values (SIMD)
- The data values are independent
- Operation use the same type of functional unit for all data
- Data values are stored in separate registers
- Data values are arranged in uniform structure (vector)
- Load/Stores access
  - a continuous range of memory
  - use a regular pattern (strided load/store)
- One instruction stream for parallel pipelines (so called lanes)

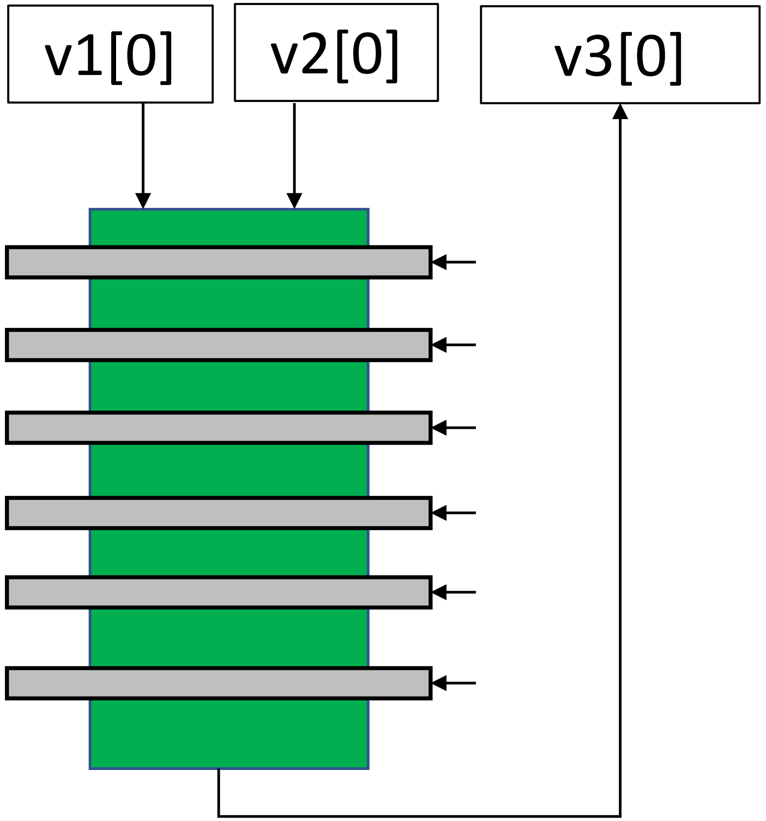
Functional Units (FUs) for Vector Arithmetic

- Input and Output are an array (vector)
  v1= [v1[0] v1[1] v1[2]… v1[n]]

- FUs operate on one element of vector
  e.g. Multiplier: v3[i] = v1[i]*v2[i]

- FUs exist for different data types
  (integer, floating point)

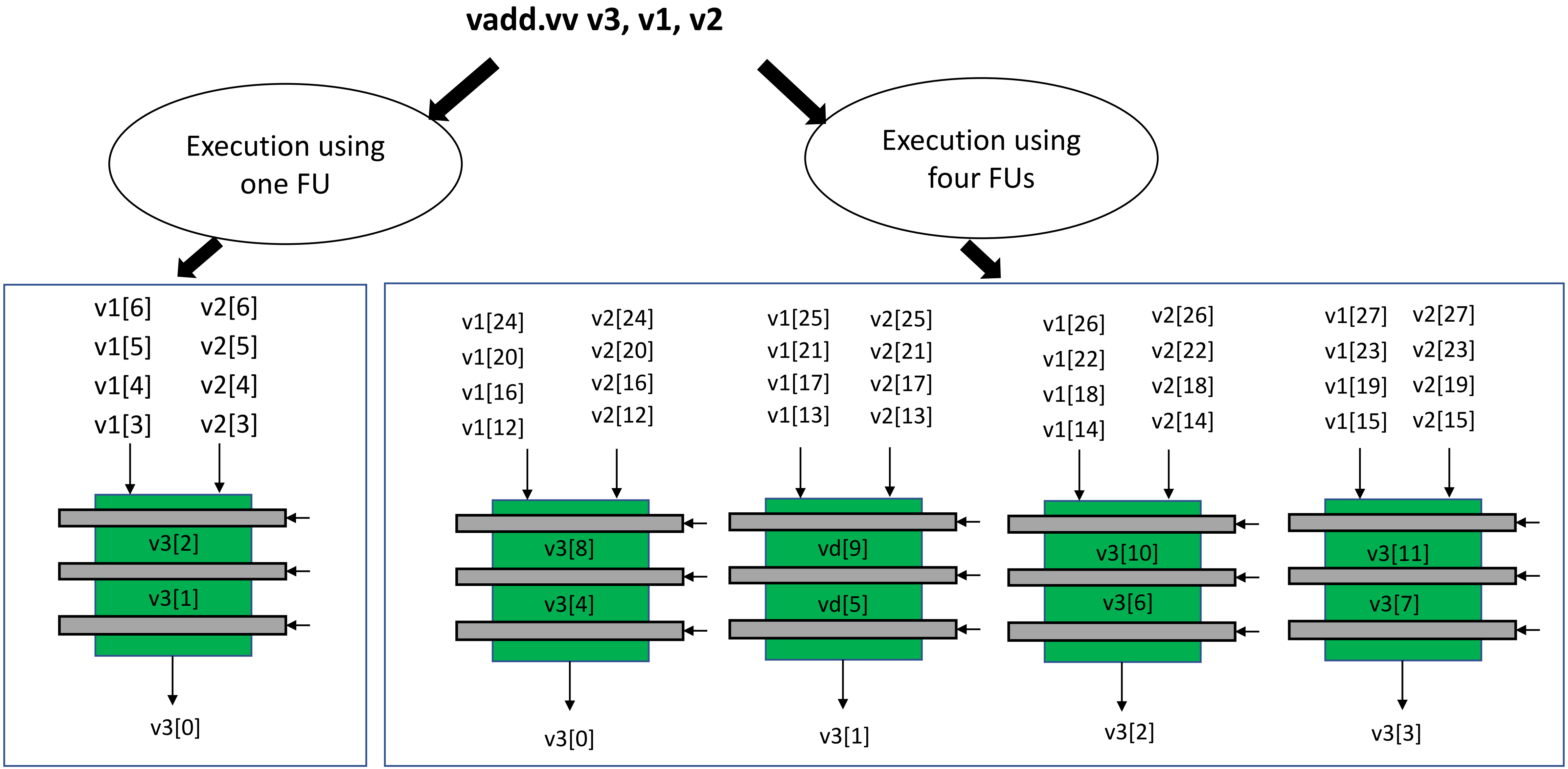
- FUs often use deep pipeline for high frequency
- Initialization Interval usually = 1
  - R: Red Operands
  - O: Operation
  - W: Wreite Result



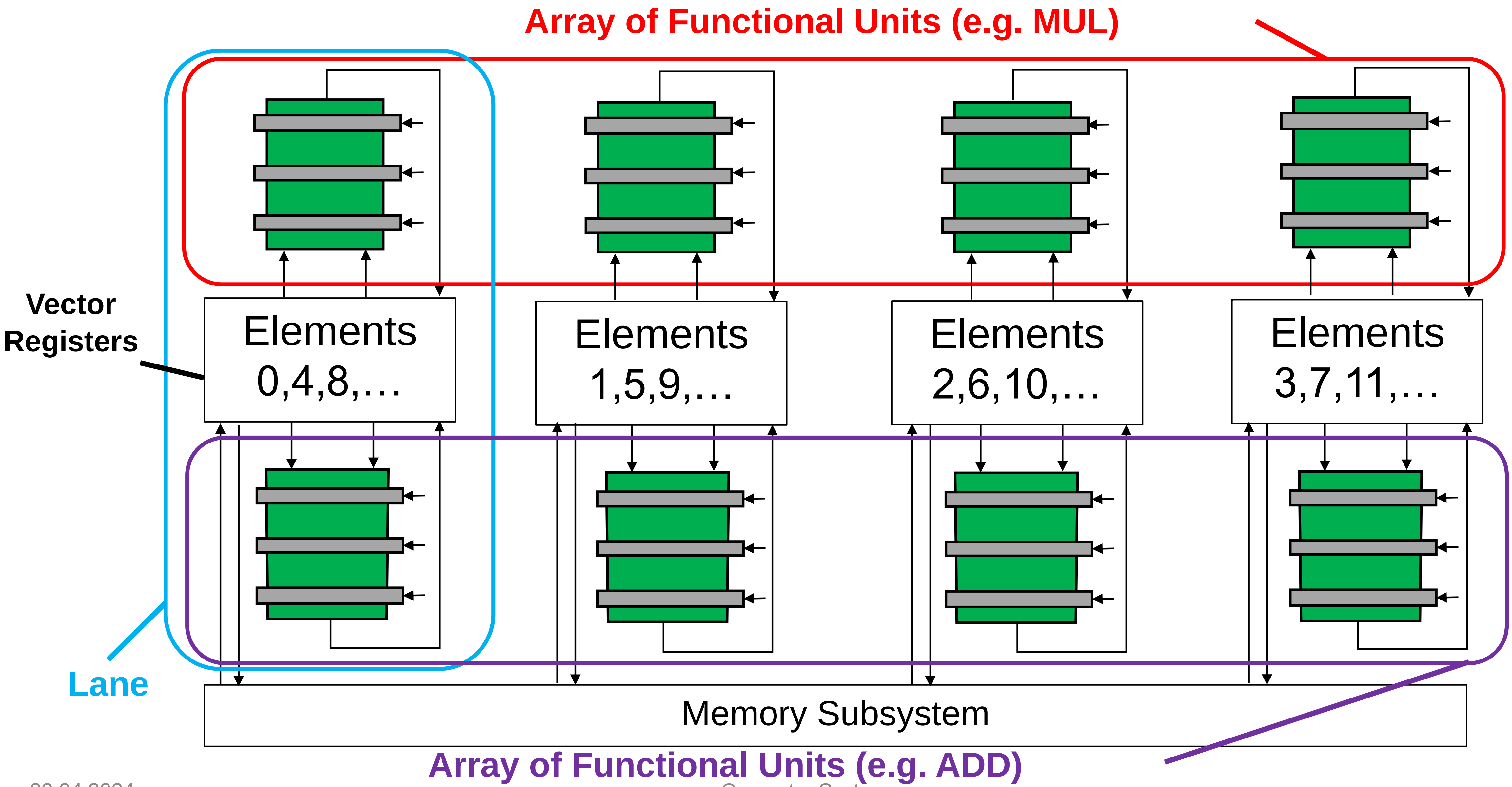
Six-stage Pipelined FU
Latency = 6

Clock Cycle	1	2	3	4	5	6
	R	O	O	O	O	W

Vector Instruction Execution on FUs



Basic Structure of a Vector Unit



Example – Timing for Single Vector Instruction

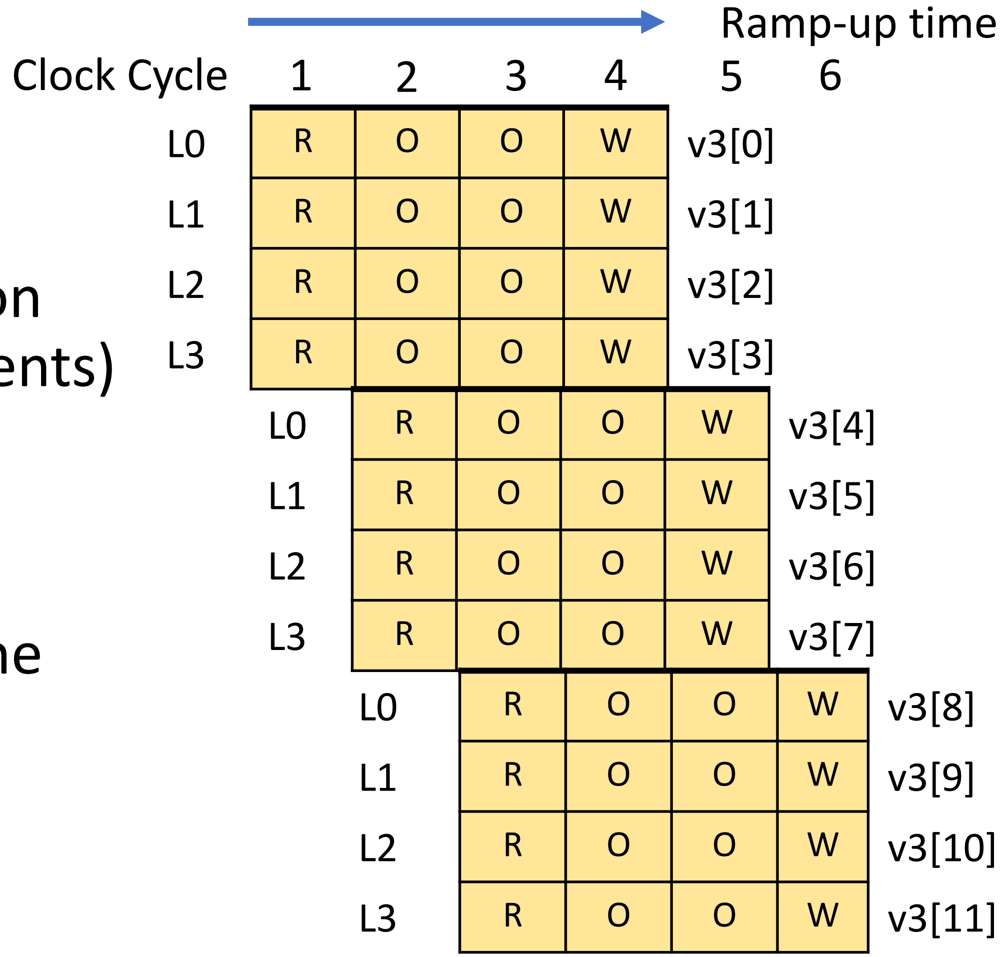
- Execution on Vector Unit
  - with four lanes (L0-L3)
  - FUs with 4 stages
  - Vector size is 12

- Lanes are used in pipelined fashion (no dependencies between elements)

- Full result is ready after 6 cycles

- 4 cycles ramp-up to fill the pipeline

```
vmul.vv v3, v1, v2
```

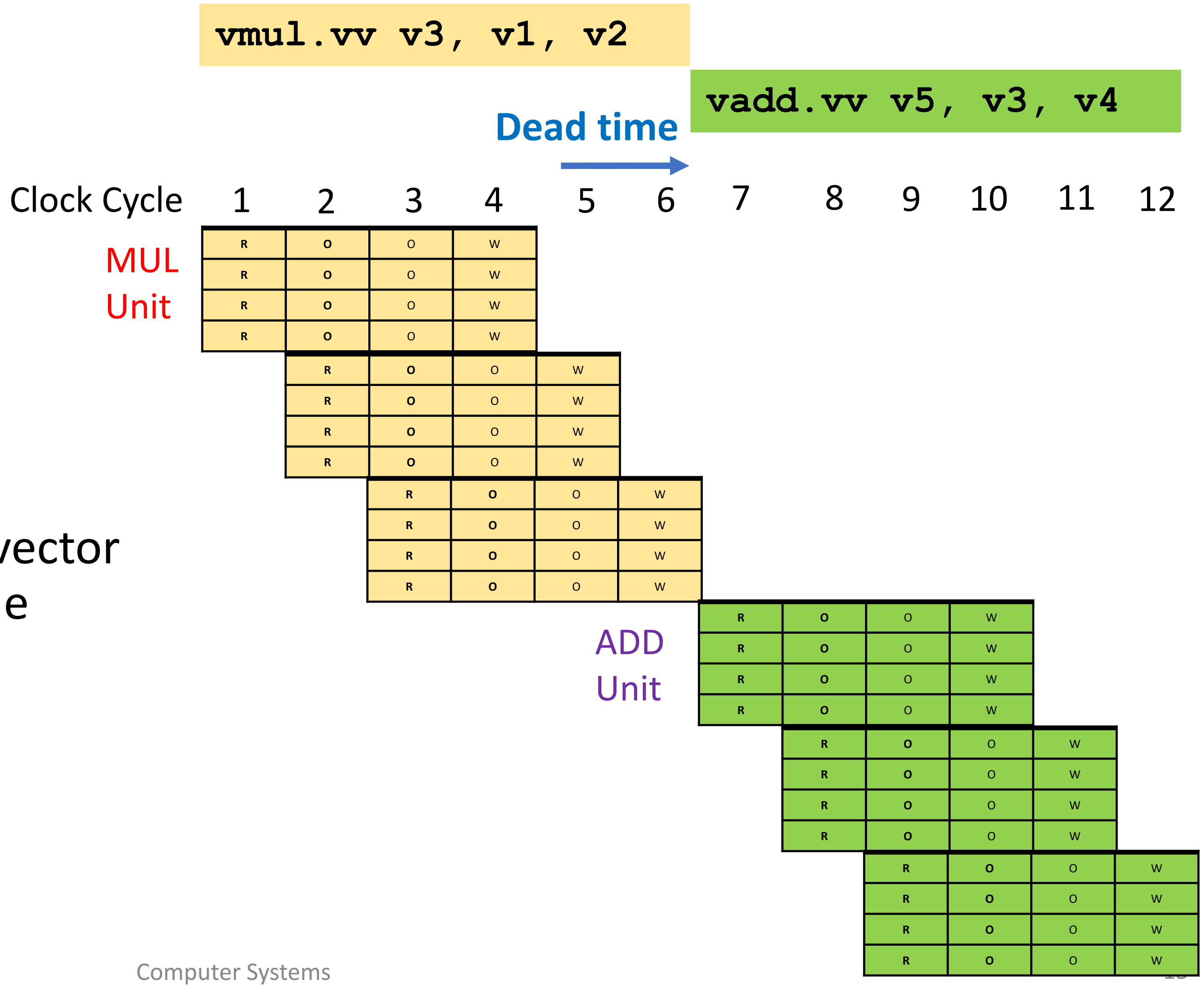


Ramp-up time

Clock Cycle	1	2	3	4	5	6	
L0	R	O	O	W			v3[0]
L1	R	O	O	W			v3[1]
L2	R	O	O	W			v3[2]
L3	R	O	O	W			v3[3]
L0		R	O	O	W		v3[4]
L1		R	O	O	W		v3[5]
L2		R	O	O	W		v3[6]
L3		R	O	O	W		v3[7]
L0			R	O	O	W	v3[8]
L1			R	O	O	W	v3[9]
L2			R	O	O	W	v3[10]
L3			R	O	O	W	v3[11]

Example – Timing for Sequence of Vector Instructions

- Full result only ready after last cycle of vector instruction
- An instruction using the result needs to wait until completed
- Causes a dead time (also called recovery time) – delay until next vector instruction can start down pipeline



```
vmul.vv v3, v1, v2
```

```
vadd.vv v5, v3, v4
```

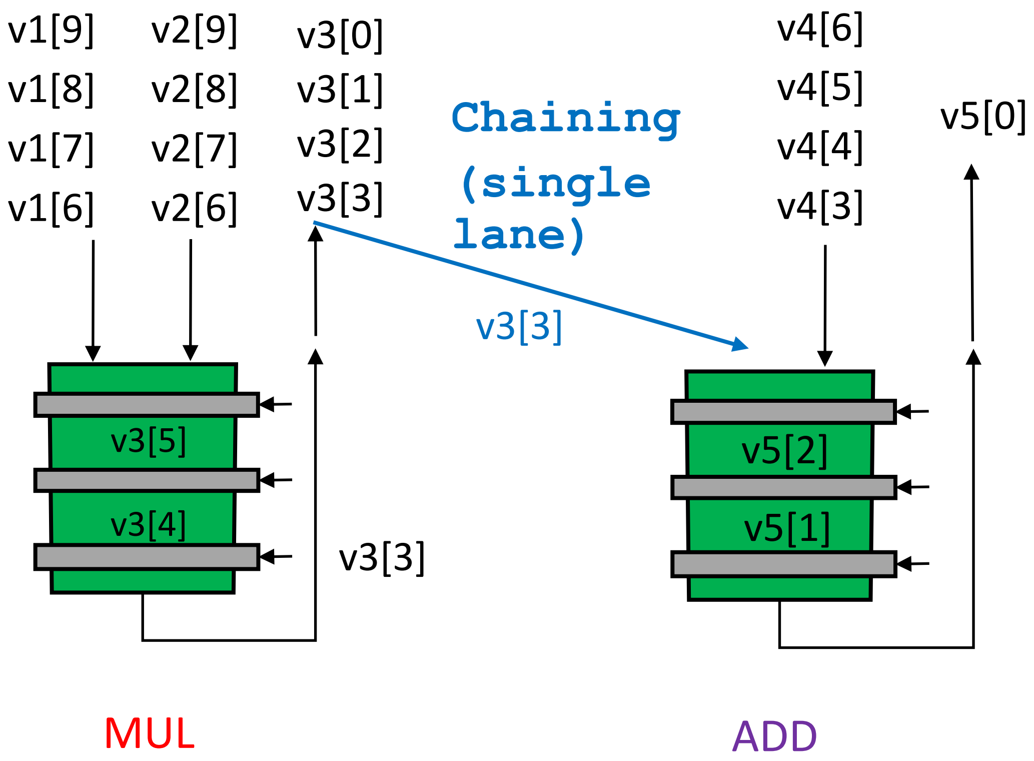
Clock Cycle	1	2	3	4	5	6	7	8	9	10	11	12
MUL Unit	R	O	O	W								
	R	O	O	W								
	R	O	O	W								
	R	O	O	W								
		R	O	O	W							
		R	O	O	W							
		R	O	O	W							
		R	O	O	W							
			R	O	O	W						
			R	O	O	W						
			R	O	O	W						
			R	O	O	W						
ADD Unit							R	O	O	W		
							R	O	O	W		
							R	O	O	W		
							R	O	O	W		
								R	O	O	W	
								R	O	O	W	
								R	O	O	W	
								R	O	O	W	
									R	O	O	W
									R	O	O	W
									R	O	O	W
									R	O	O	W

Vector Chaining

- Vector version of forwarding paths
- Results are forwarded element-wise to next FU via chaining

```
vmul.vv    v3, v1, v2

vadd.vv    v5, v3, v4
```

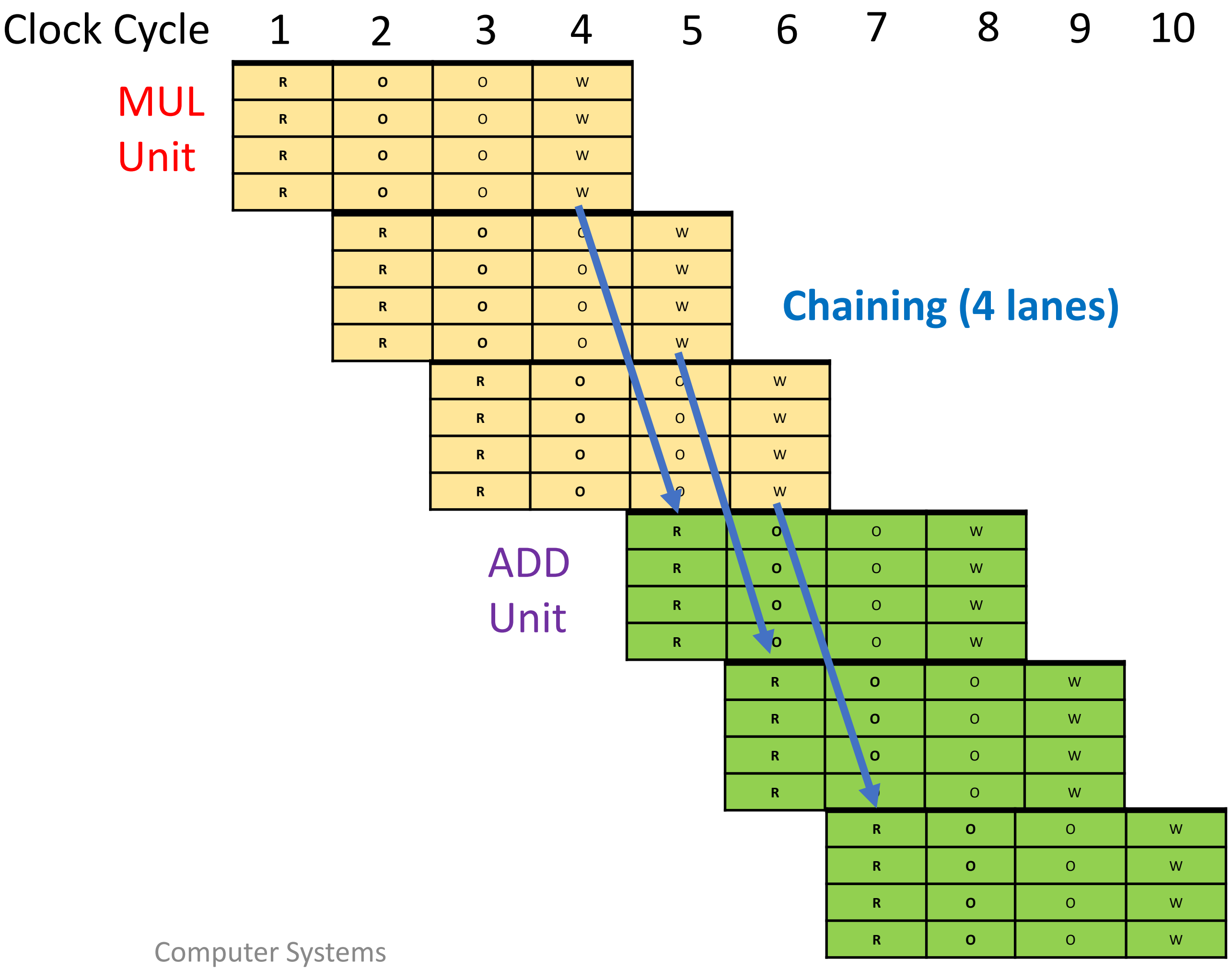


Example – Timing for Sequence of Vector Instructions with Chaining

- Chain: Forward results from all lanes between the FUs
- No dead time

```
vmul.vv v3, v1, v2
```

```
vadd.vv v5, v3, v4
```



Clock Cycle	1	2	3	4	5	6	7	8	9	10	11	12
MUL Unit	R	O	O	W								
	R	O	O	W								
	R	O	O	W								
	R	O	O	W								
		R	O	O	W							
		R	O	O	W							
		R	O	O	W							
		R	O	O	W							
			R	O	O	W						
			R	O	O	W						
			R	O	O	W						
			R	O	O	W						
ADD Unit					R	O	O	W				
					R	O	O	W				
					R	O	O	W				
					R	O	O	W				
						R	O	O	W			
						R	O	O	W			
						R	O	O	W			
						R	O	O	W			
							R	O	O	W		
							R	O	O	W		
							R	O	O	W		
							R	O	O	W		

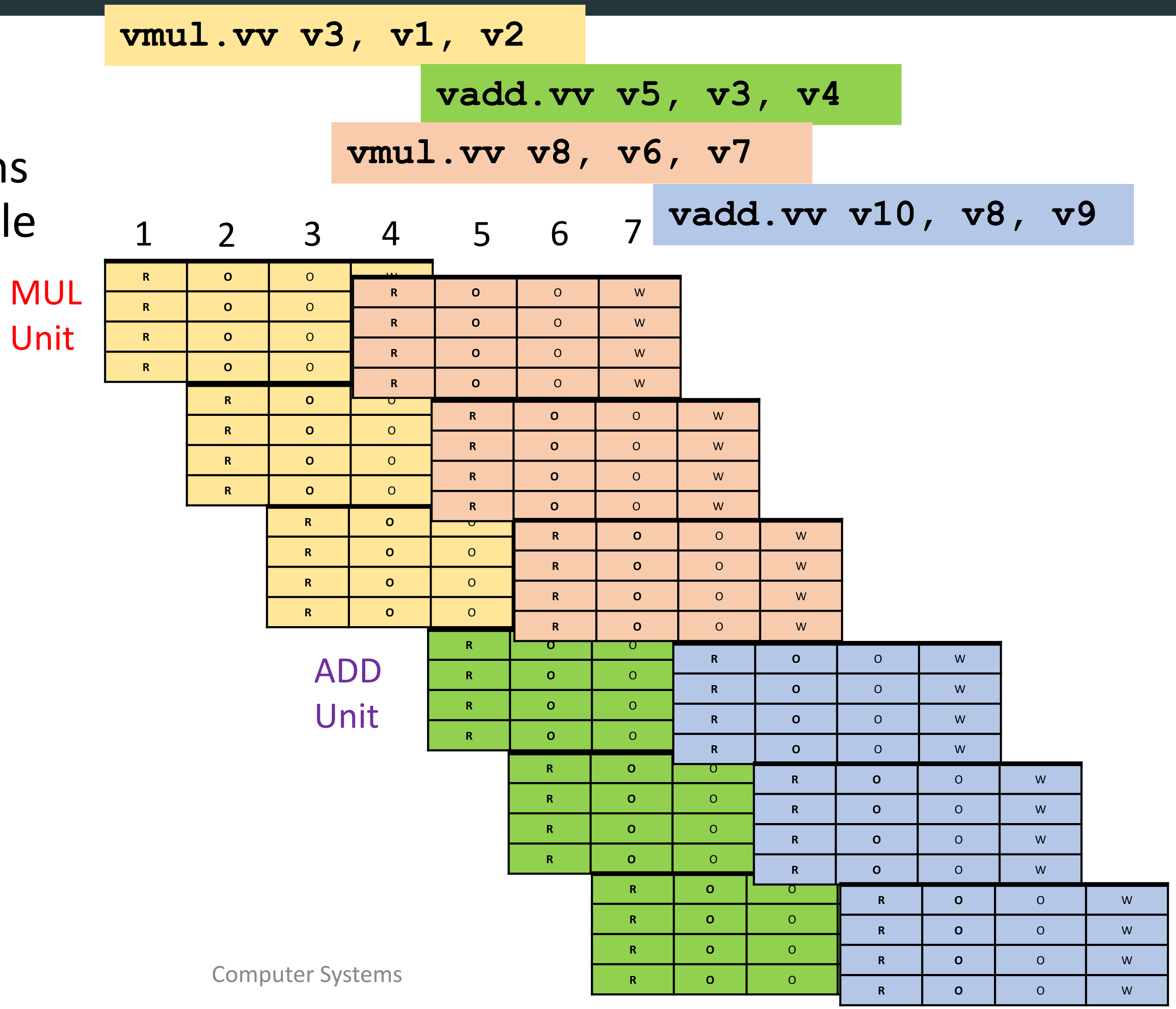
Example – Timing for Sequence of Vector Instructions with Chaining and Interleaving

- Interleaving can overlap independent vector instructions as soon as FUs become available

- Example:

```
vmul.vv    v3, v1, v2
vadd.vv    v5, v3, v4

vmul.vv    v8, v6, v7
vadd.vv    v10, v8, v9
```



The RISC-V Vector Instruction Set

RISC-V Vector Programming Model

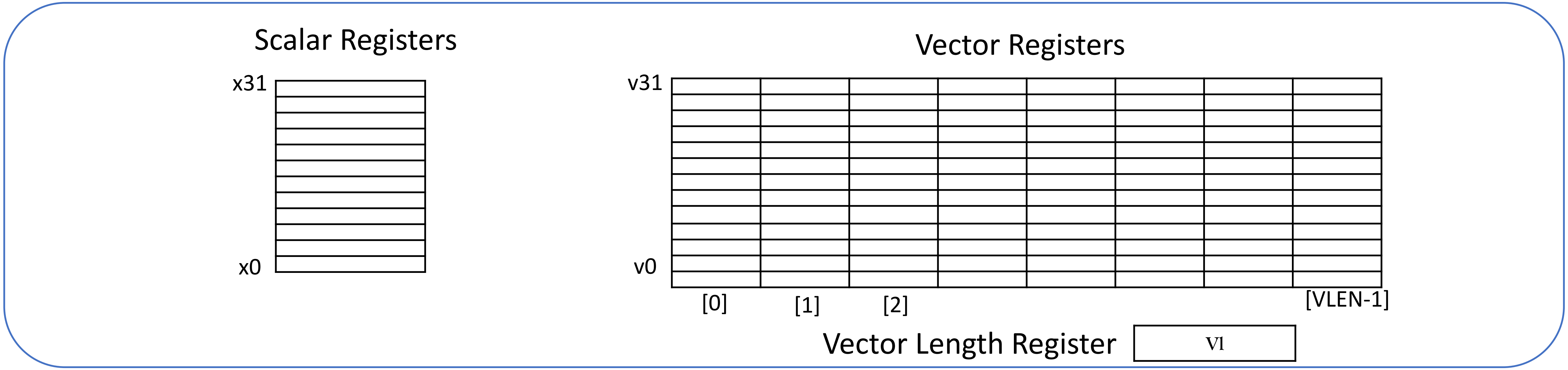
- RISC-V “V” Vector Extension
  - Standard extension to the RISC-V ISA
  - Version 1.0: https://github.com/riscv/riscv-v-spec

- Memory-register vector instructions (operations on registers)
- Vector and vector element sizes are configurable
  (Vectors can be longer than one vector register)

- CSR: Specialized registers to save configuration and status of processor

RISC-V Vector Programming Model

- Vector Registers and vector length
  - 32 vector data registers (v0~v31) : each ***VLEN*** bits long
  - Vector length register ***vl***
    - defines on how many elements will the next vector operations be executed
  - Vector type register ***vtype*** *(see next slide)*

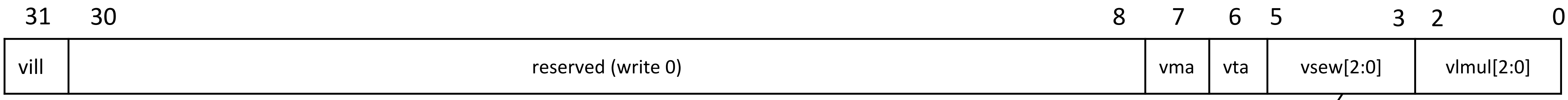


Vector CSRs

- Vector type register: **vtype**
  - Used to define vector length via parameters **SEW** and **LMUL**
  - Used to define tail and mask policy via **vta** and **vma**
  - See next slide for details
- Vector Byte Length: **vlenb**
  - Read-only; Holds the value **VLEN/8** (a design-time constant)
  - Used to define the vector register length **VLEN** (fixed)
- Vector Length Register: **vl**
  - Read-only; Can be updated by the **vset{i}vl{i}** instructions (see slide 25)
  - Used to define on how many elements will the next vector operations be executed
  - ***vl*** is limited by ***VLMAX=LMUL * VLEN / SEW***
- Vector Start Index: **vstart**
  - Used to define the index of first element to be executed by a vector instruction

CSR: Vector type register vtype

- CSR: Vector type register **vtype** layout:



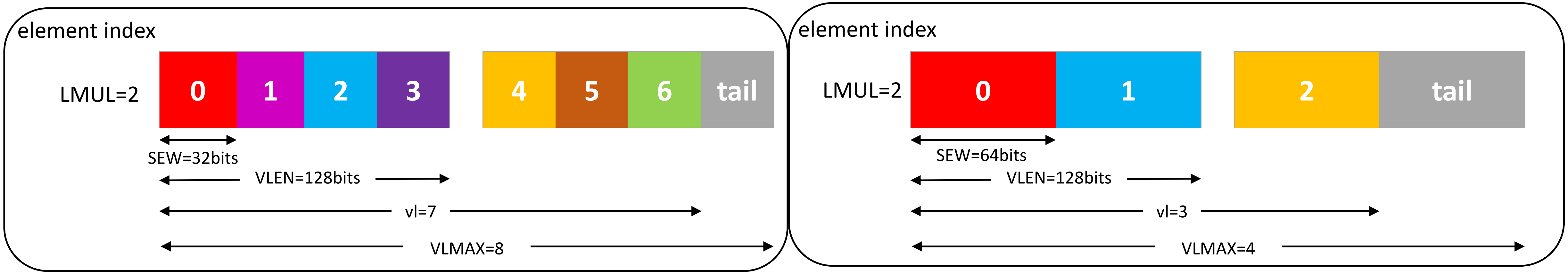
31	30 … 8	7	6	5 … 3	2 … 0
vill	reserved (write 0)	vma	vta	vsew[2:0]	vlmul[2:0]

- ***vsew[2:0]*** field encodes selected element width(SEW)
  - the elementary size (in bits) of an element in the input and output vectors
  - SEW = 8 $*2^{vsew}$
- ***vlmul[2:0]*** encodes vector register length multiplier
  - LMUL = 2^{vlmul} = 1/8 … 8
  - defines the size of the vector group for vector operation input and output operands, that is the number of vector register(s) forming the group
- ***vta*** specifies *tail-agnostic/tail-undisturbed* policy
- ***vma*** specifies *mask-agnostic/mask-undisturbed* policy

vsew[2:0]			SEW
0	0	0	8
0	0	1	16
0	1	1	32
1	0	0	64
1	0	1	128
1	0	1	256
1	1	0	512
1	1	1	1024

RISC-V Vector Programming Model - Vector Layouts

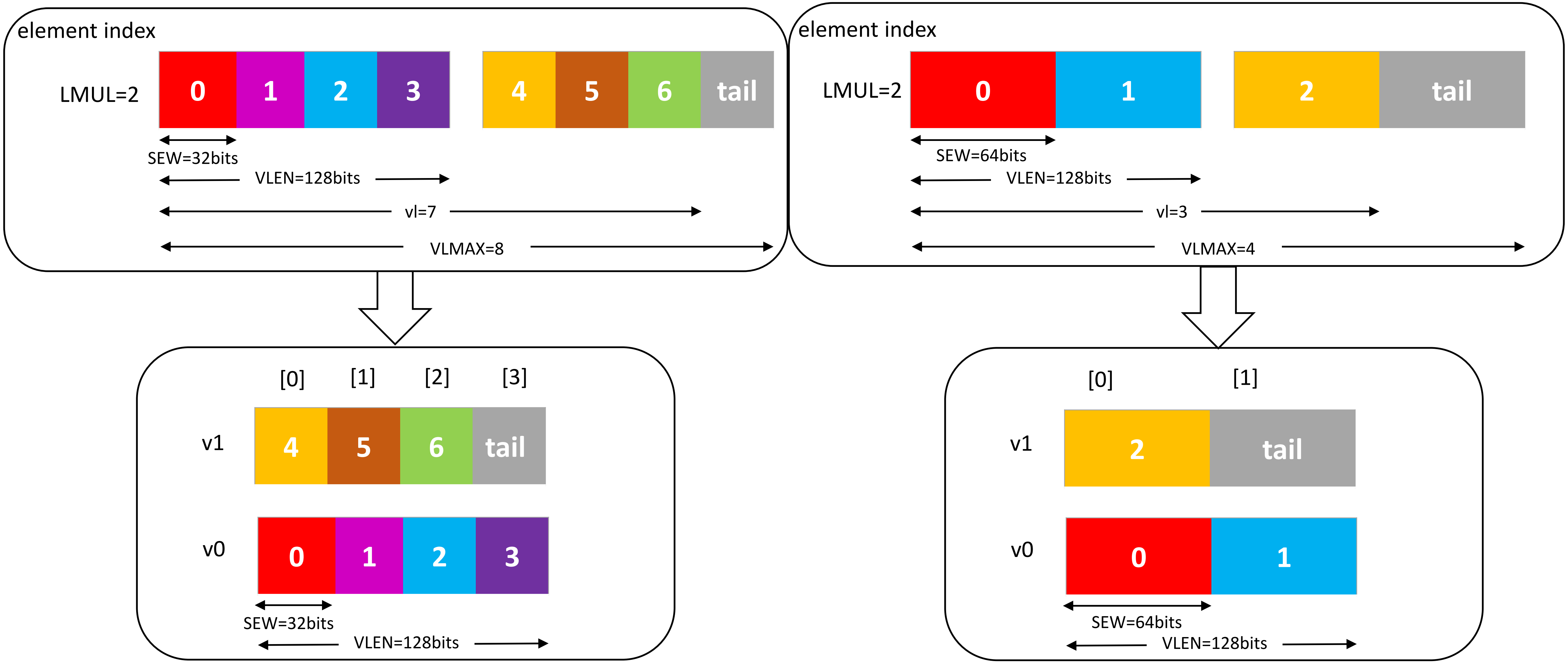
- Example vector register data layouts



- ***vl*** is limited by ***VLMAX=LMUL * VLEN / SEW***
- Tail : the elements past the vector length ***vl***; not affected by the current operation
- Two tail policies: *undisturbed* & *agnostic*
  - *undisturbed* : the tail elements are left unmodified
  - agnostic : the tail elements are left undisturbed or fill in with all 1s

RISC-V Vector Programming Model - Vector Layouts in Vector Registers

- Example vector register data layouts

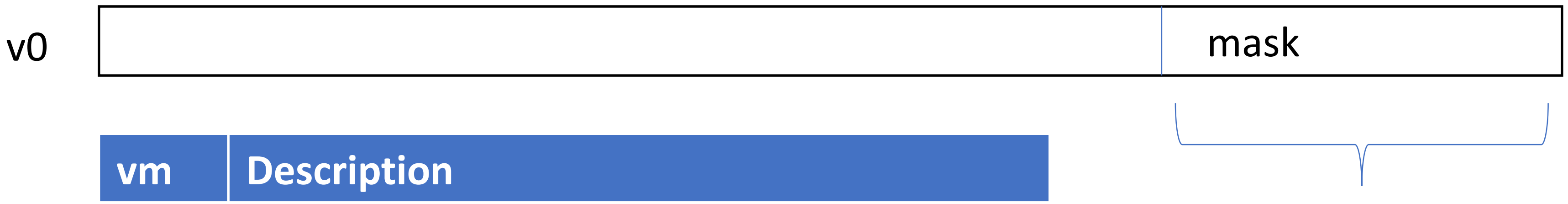


Vector Masking

- The mask value used to control execution of a masked vector instruction is always supplied by vector register **v0**
- Where available, masking is encoded in a **single-bit vm** field in the instruction word

- vadd.vv vd, vs2, vs1   #unmasked vector operation, vm=1 in instruction

  vadd.vv vd, vs2, vs1, **v0.t**   #enabled masking, mask supplied in v0, vm=0 in instruction



vm	Description
0	vector result, only where v0.mask[i] = 1
1	unmasked

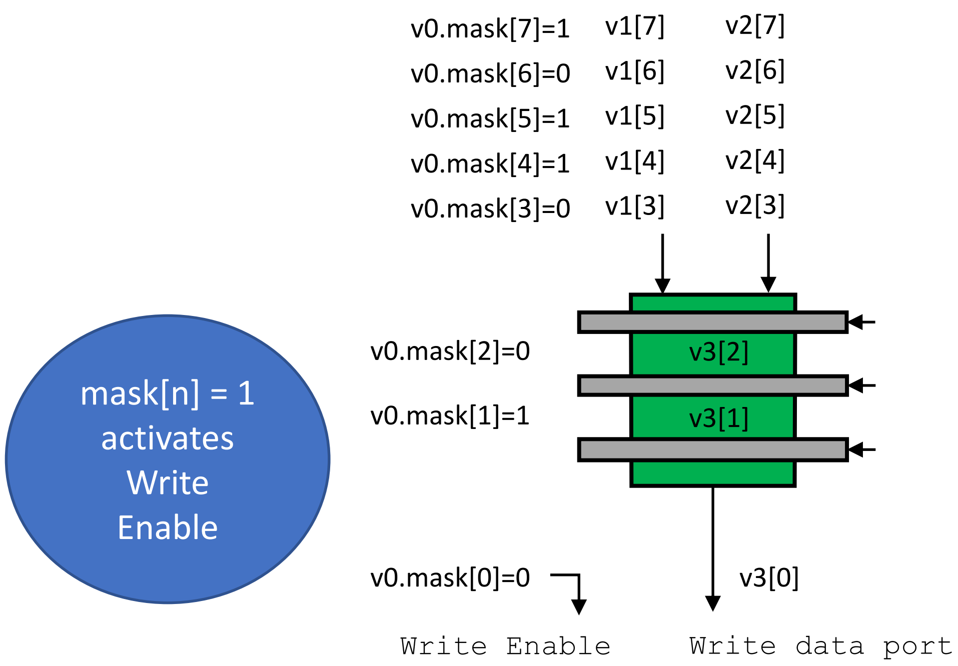
n-th mask bit v0.mask[n] is the
n-th bit of the vector register v0

RISC-V Vector Programming Model

- Masking
  - This bitmask defines which of the result element should be actually modified by the operation
  - Two mask policies : *undisturbed* & *agnostic*
    - *undisturbed* : mask-off elements keep the value they had before the operation
    - *agnostic* : mask-off elements can either be undisturbed or written with all 1s.

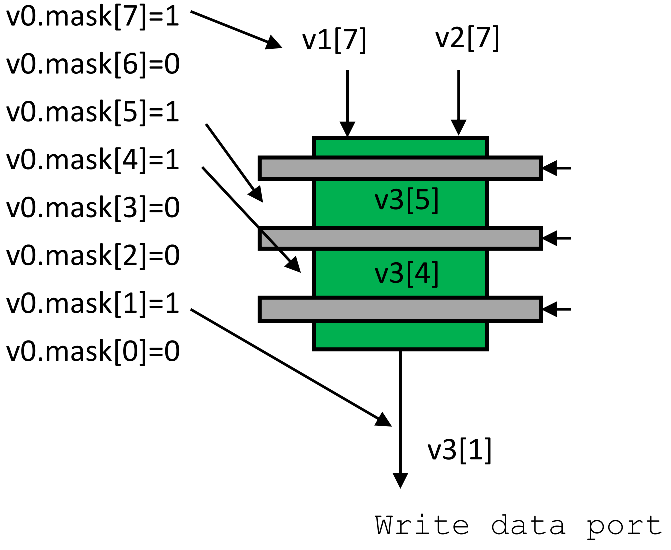
Simple Implementation

Execute all N operations, turn off result writeback according to mask



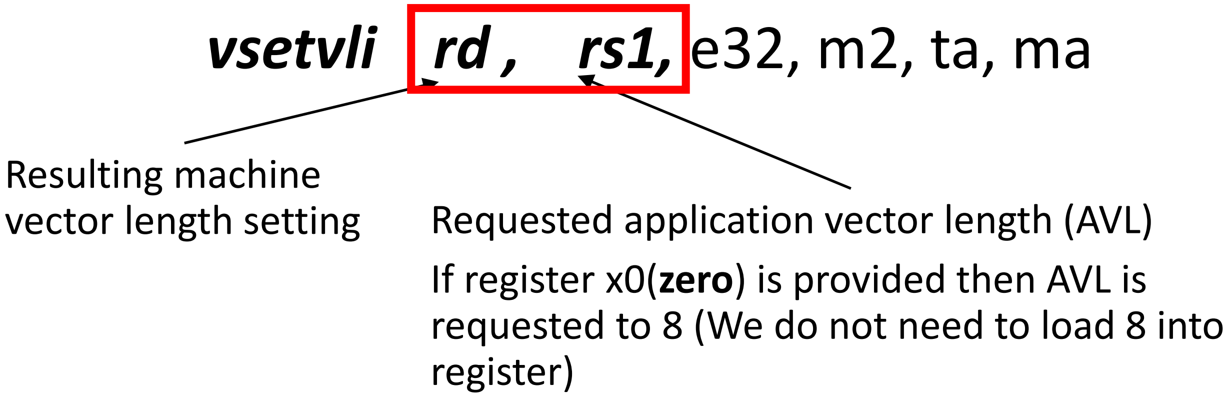
Density-Time Implementation

Scan mask vector and only execute elements with non-zero masks



RISC-V Vector Programming Model

- Setting the vector configuration via ***vsetvli***
- The ***vsetvli*** configuration instructions set the ***vtype*** register, and also set the vl register, returning the vl value in a scalar register



vsetvli ***rd ,*** ***rs1,*** e32, m2, ta, ma

Resulting machine vector length setting

Requested application vector length (AVL)

If register x0(**zero**) is provided then AVL is requested to 8 (We do not need to load 8 into register)

- Resulting machine vector length in rd: vl = min (LMUL*VLEN / SEW , rs1)

RISC-V Vector Programming Model

- Setting vector configuration, *vsetvli*
- The ***vsetvli*** configuration instructions set the ***vtype*** register, and also set the vl register, returning the vl value in a scalar register

vsetvli rd , rs1, e32, m2, ta, ma

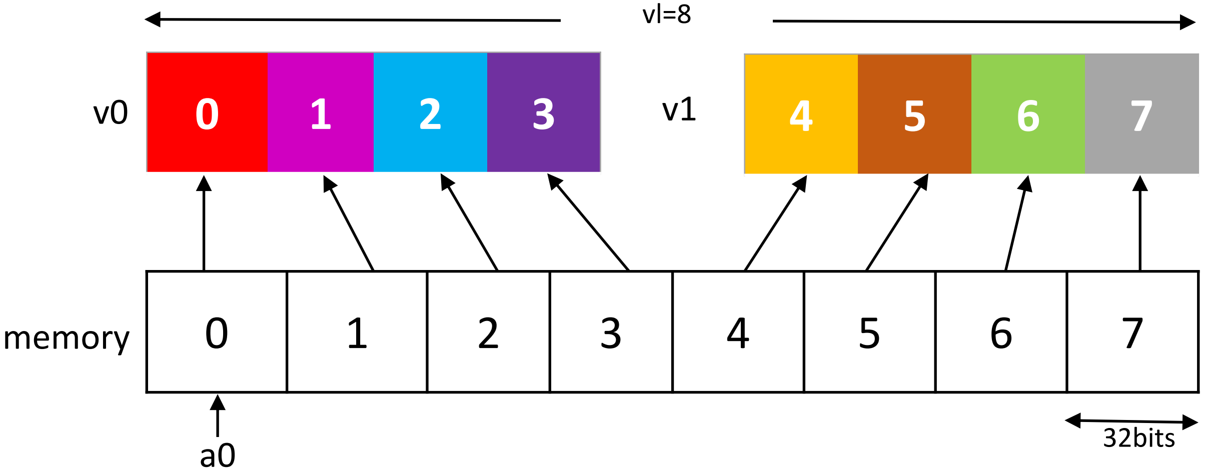
vtype parameters (SEW, LMUL, VTA, VMA)
encoded as immediate in instruction

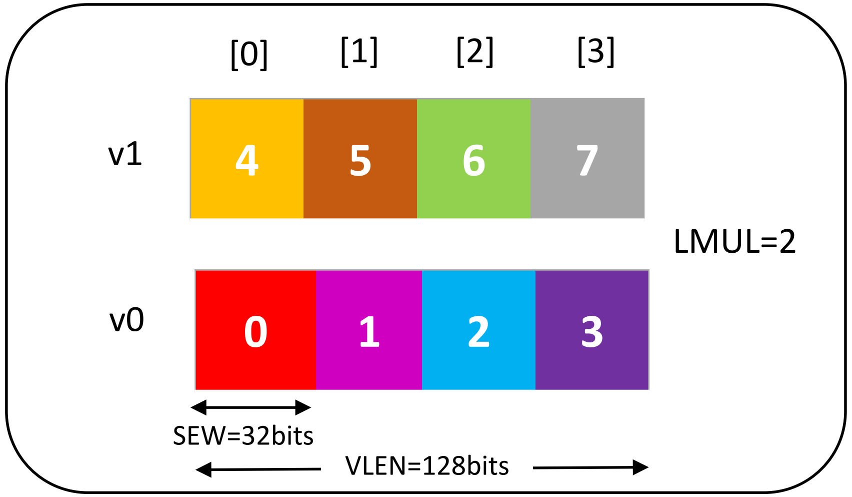
e8	#SEW = 8bits
e16	#SEW = 16bits
e32	**#SEW = 32bits**
e64	#SEW = 64bits
mf8	#LMUL =1/8
mf4	#LMUL = 1/4
mf2	#LMUL =1/2
m1	#LMUL = 1, default
m2	**# LMUL =2**
m4	#LMUL=4
m8	#LMUL=8
tu	#tail undisturbed, default
ta	**#tail agnostic**
mu	#mask undisturbed, default
ma	**#mask agnostic**

RISC-V Vector Programming Model

- Vector Load and Store
  - If set VLEN=128 & vsetvli t0, zero ,e32, m2, ta, ma
  - vl2re32.v   v0, (a0)

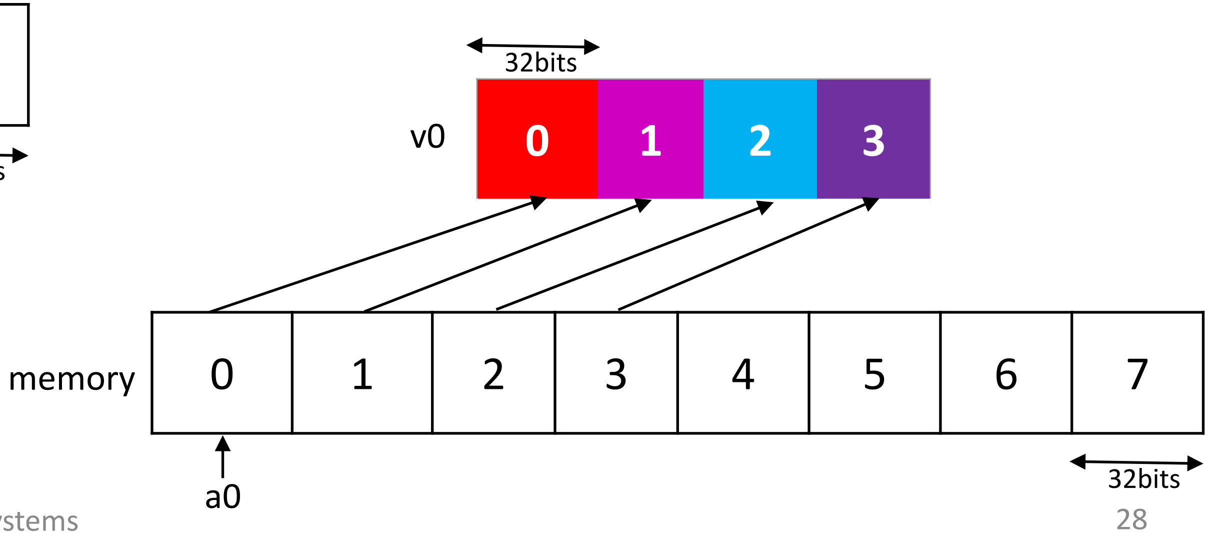
# Load v0-v1 with 2*VLEN/32 words(32bits) held at address in a0





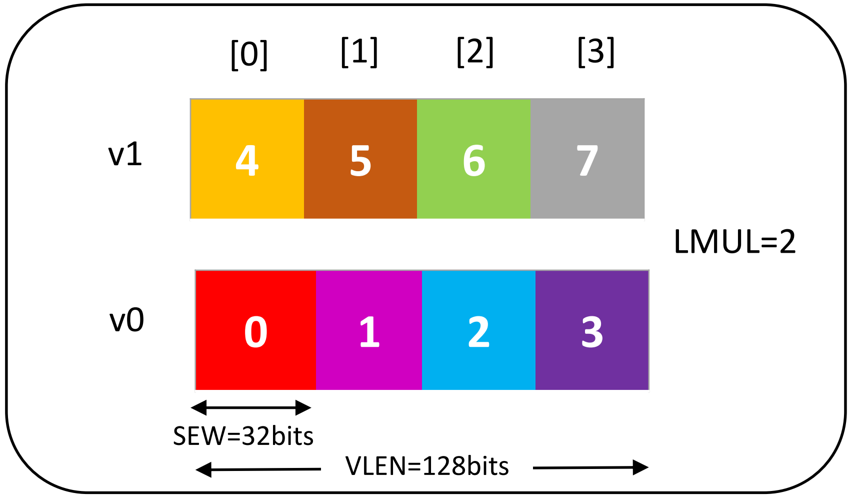
vle32.v v0 , (a0)

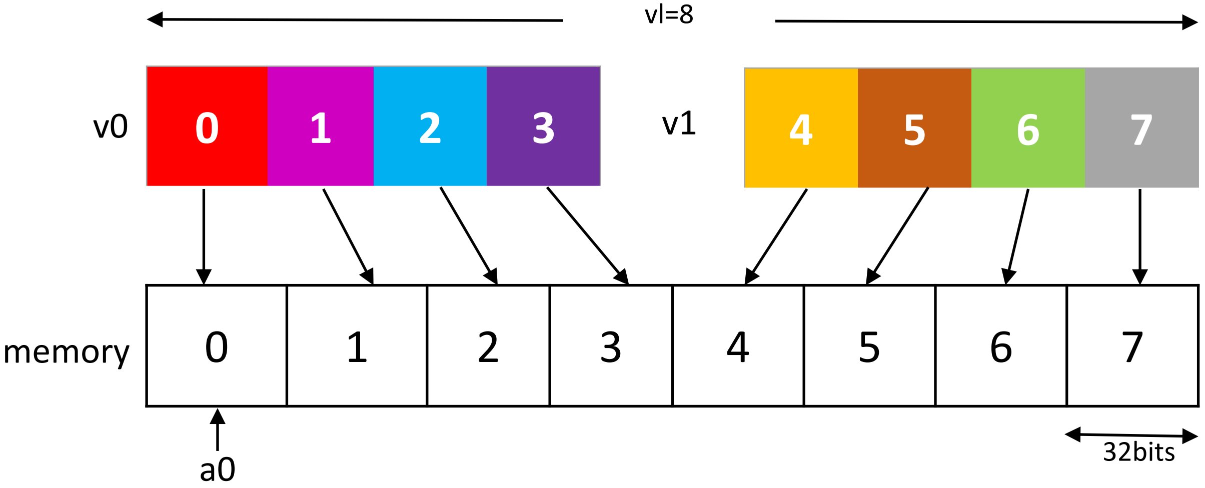
# 32-bit unit-stride load



RISC-V Vector Programming Model

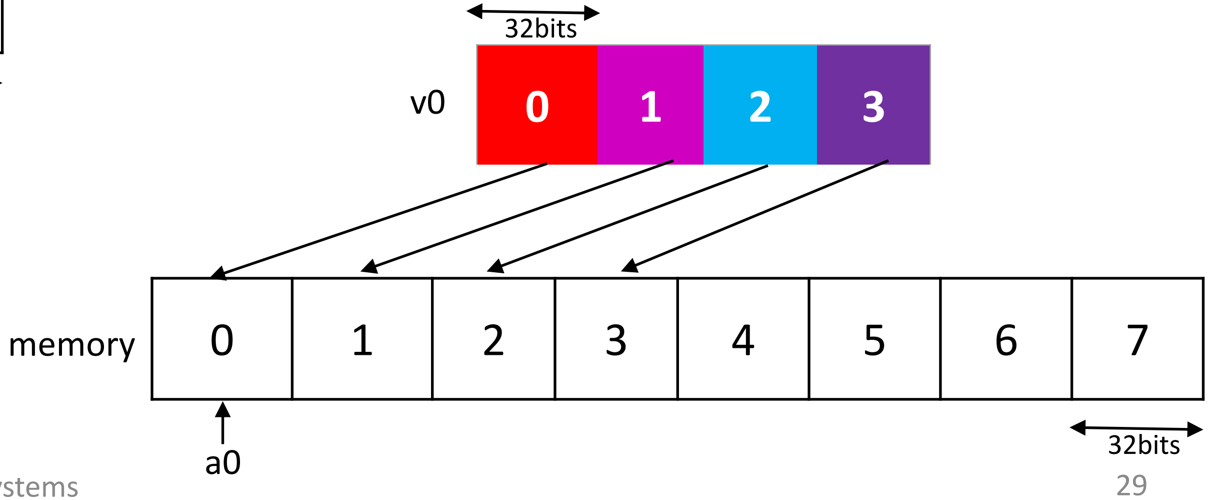
- Vector Load and Store:
  - If set VLEN=128 & vsetvli t0, zero ,e32, m2, ta, ma
  - vs2r.v v0, (a0)   # Store v0-v1 to address in a0





vse32.v v0 , (a0)

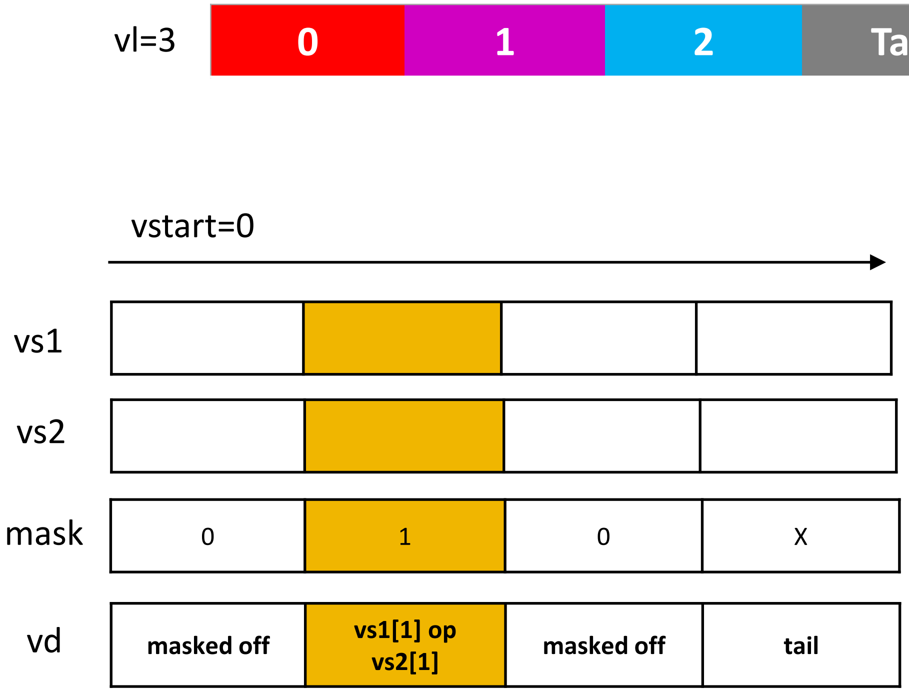
# 32-bit unit-stride store



RISC-V Vector Programming Model

vstart :

- specifies the first active vector element
- vstart is also saved in a CSR



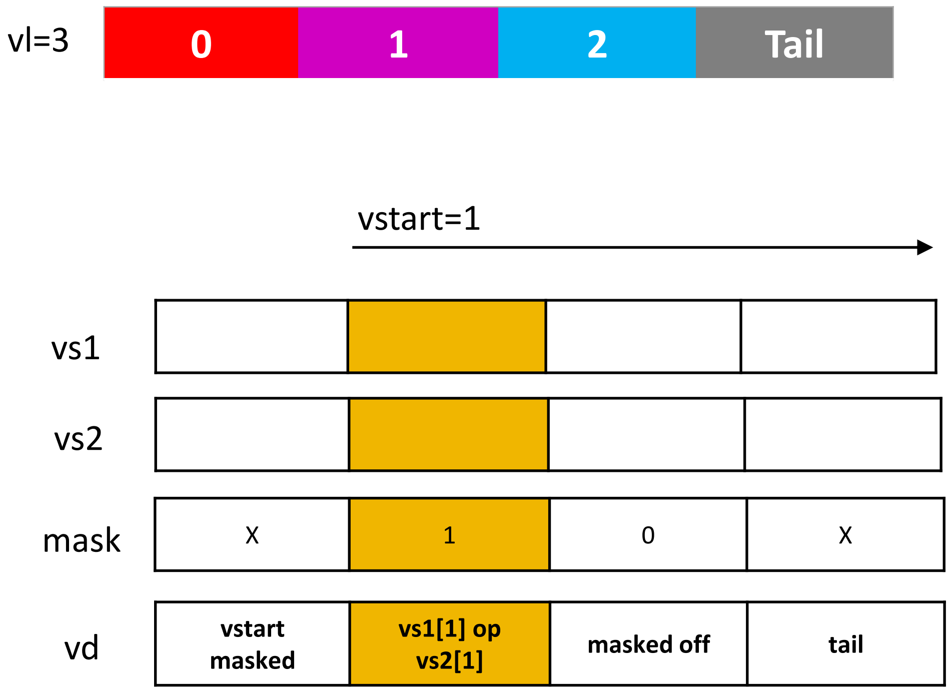
vl=3

0	1	2	Tail

vstart=0

vs1				
vs2				
mask	0	1	0	X
vd	masked off	vs1[1] op vs2[1]	masked off	tail

Masked-off and tail elements follow mask and tail policies : which are parameters defined in the vtype CSR register



vl=3

0	1	2	Tail

vstart=1

vs1				
vs2				
mask	X	1	0	X
vd	vstart masked	vs1[1] op vs2[1]	masked off	tail

Masked-off and tail elements follow mask and tail policies : which are parameters defined in the vtype CSR register

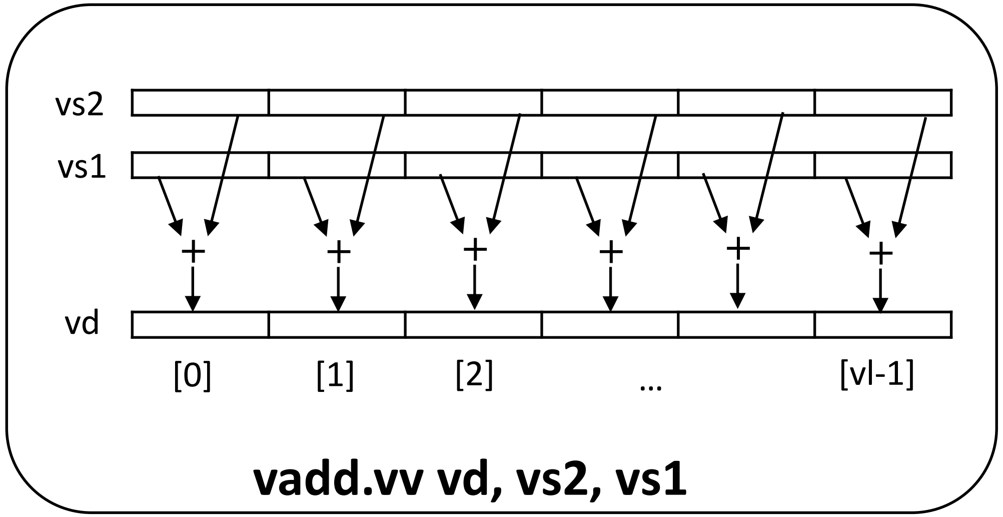
RISC-V Vector Programming Model

Vector-Vector Operation:

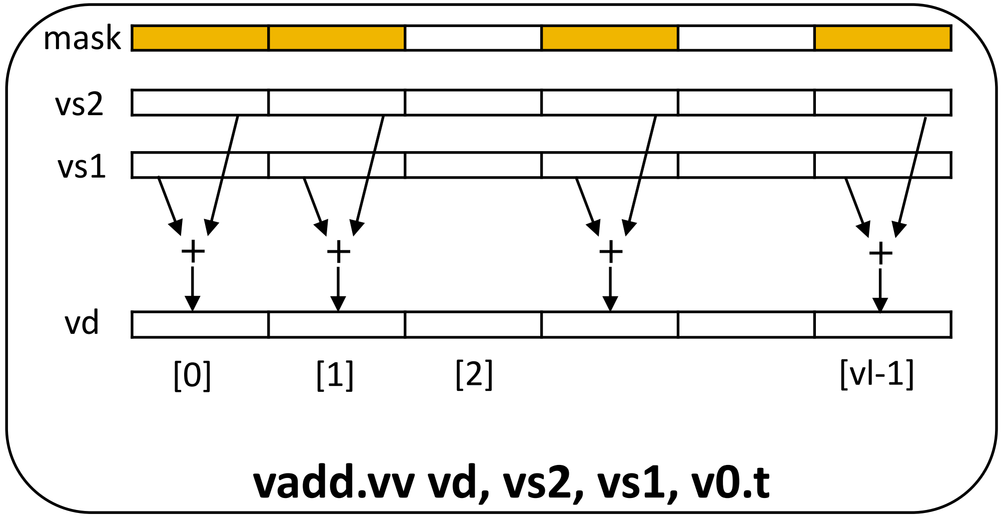
- Addition: vadd.vv
- Multiplication: vmul.vv

Operation is conducted element-wise between the two vectors.

Without/with masking.



vadd.vv vd, vs2, vs1

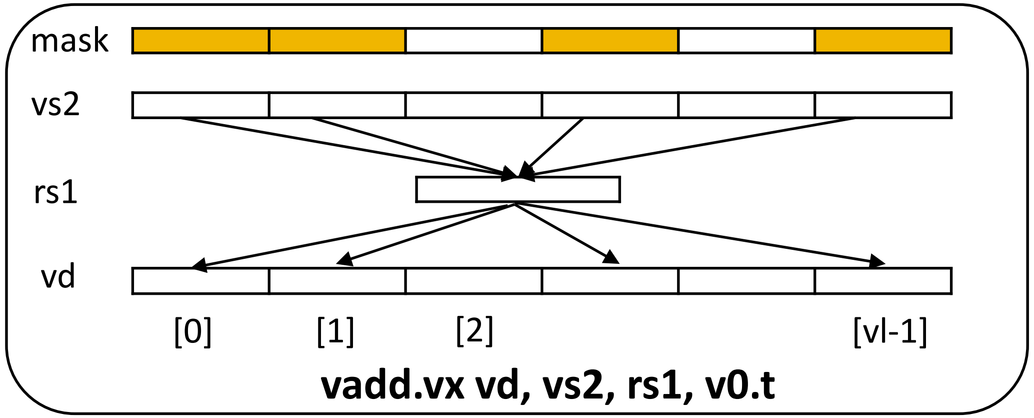


vadd.vv vd, vs2, vs1, v0.t

RISC-V Vector Programming Model

Vector-Scalar Operation:

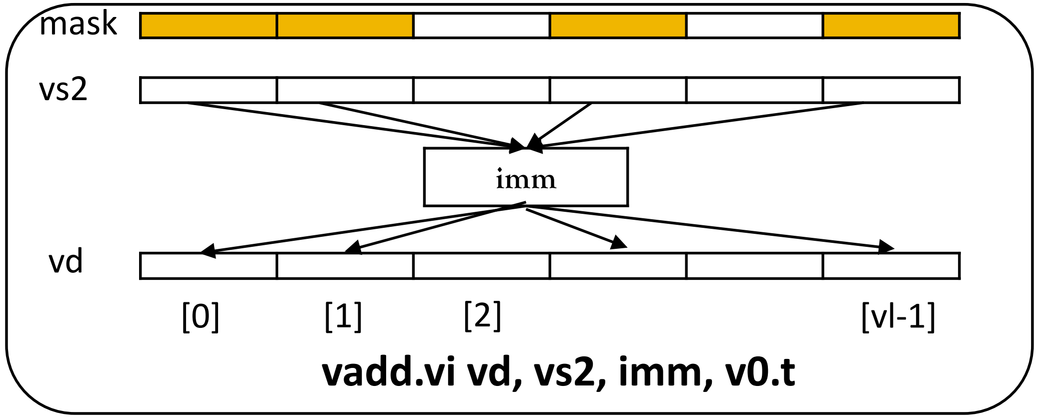
Operation is conducted between each unmasked element of the vector and a scalar register value.



vadd.vx vd, vs2, rs1, v0.t

Vector Immediate Operation:

Operation is conducted between each unmasked element of the vector and a constant value.



vadd.vi vd, vs2, imm, v0.t

Vector Code Example

```
# C code
for ( i = 0; i < 8; i++)
      C[i] = A[i] + B[i];
```

```
# Scalar Code
    li    a0,   8
loop:
    lw  a4, 0(a1)
    lw  a5, 0(a2)
    add a4, a4, a5
    sw   a4, 0(a3)
    addi a3, a3, 4
    addi a2, a2, 4
    addi a1, a1, 4
    addi a0, a0, -1
    bnez a0, loop
```

```
# Vector Code
vsetvli  t0, zero ,e32, m2,
ta, ma   # t0 = 8
vl2re32.v   v8, (a1)
vl2re32.v   v10, (a2)
vadd.vv      v8, v10,v8
vs2r.v          v8, (a3)


# (a1)     A
# (a2)     B
# (a3)     C
```

Vector Code Example

```
# C code
for ( i = 0; i < 8 ; i++)
      C[i] = A[i] * B[i];
```

```
# Scalar Code
     li   a0, 8
loop:
    lw  a4, 0(a1)
    lw  a5, 0(a2)
    mul a4, a5, a4
    sw   a4, 0(a3)
    addi a3, a3, 4
    addi a2, a2, 4
    addi a1, a1, 4
    addi a0, a0, -1
    bnez a0, loop
```

```
# Vector Code
vsetvli  t0, zero ,e32, m2,
ta, ma   # t0 = 8
vl2re32.v   v8, (a1)
vl2re32.v   v10, (a2)
vmul.vv      v8, v10,v8
vs2r.v          v8, (a3)


# (a1)     A
# (a2)     B
# (a3)     C
```

Vector Code Example

# C code	# Scalar Code	# Vector Code

for(i=0; i<8; i++)
        y[i]=a*x[i]+y[i];
	li a0, 8
	loop:
	      lw a4, 0(a2)
	      lw a5, 0(a3)
	      mul a4, a4, a1
	      add a4, a4 ,a5
	      sw a4, 0(a3)
	      addi a0, a0, -1
	      addi a3, a3, 4
	      addi a2, a2, 4
	      bnez a0, loop
		vsetvli t0, zero, e32, m2,
		ta, ma *# t0 = 8*
		vl2re32.v v8, (a2)
		vl2re32.v v10, (a3)
		vmacc.vx v10, a1, v8
		vs2r.v v10, (a3)
		
		# (a2) x
		# (a3) y
		# a1 a

Vector Code Example

```
# C code
#set mask
for(i=0;i<8;i++)
        mask[i] = i % 2;
for(i=0; i<8; i++){
        if(mask[i])
                y[i]=a*x[i]+y[i];
         }
```

```
# Scalar Code
      li        a0, 8
loop:
      lw      a4, 0(a2)
      lw      a5, 0(a3)
      lw      t1,  0(a4)
      beqz t1,  skip #if mask[i]=0
      mul  a4, a4, a1
      add  a4, a4 ,a5
      sw    a4, 0(a3)
skip:
      addi a0,  a0, -1
      addi a3,  a3,  4
      addi a2,  a2,  4
      bnez a0, loop
```

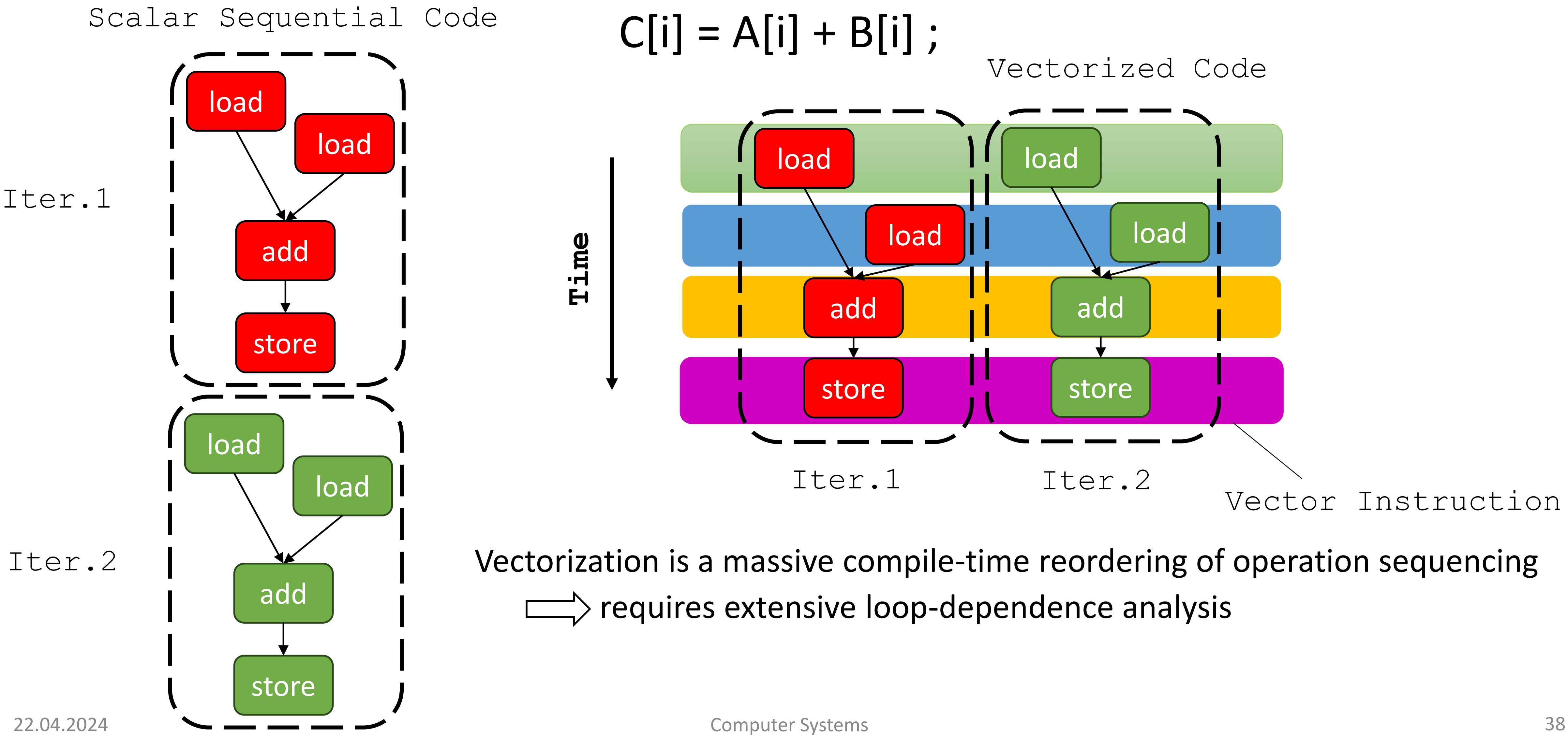
```
# Vector Code
vsetvli  t0, zero, e32, m2,
ta, ma   # t0 = 8
vl2re32.v  v8,   (a2)
vl2re32.v  v10, (a3)
vl2re32.v  v12,   (a4)
vmsne.vx  v0, v12, zero
# Set the v0, enabling the
mask if mask[i] is not zero
vmacc.vx  v10, a1,  v8, v0.t
vs2r.v        v10,  (a3)

# (a2)    x
# (a3)    y
# (a4)    mask
# a1       a
```

Vectorization

Automatic Code Vectorization

```
for (i=0; i<N; i++)
    C[i] = A[i] + B[i] ;
```



Vectorization is a massive compile-time reordering of operation sequencing

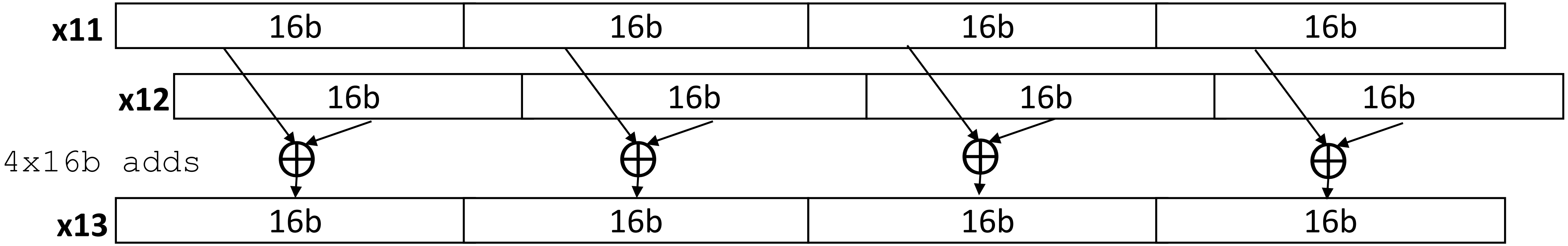
⇨ requires extensive loop-dependence analysis

Packed SIMD

Packed SIMD Extensions

<table>
<tr><td colspan="8">64b</td></tr>
<tr><td colspan="4">32b</td><td colspan="4">32b</td></tr>
<tr><td colspan="2">16b</td><td colspan="2">16b</td><td colspan="2">16b</td><td colspan="2">16b</td></tr>
<tr><td>8b</td><td>8b</td><td>8b</td><td>8b</td><td>8b</td><td>8b</td><td>8b</td><td>8b</td></tr>
</table>

- Very short vectors added to existing ISAs for microprocessors
- Use existing (32) 64-bit registers split into 2x32b or (2x16b) 4x16b or (4x8b) 8x8b
- Single instruction operates on all elements within register
- Examples:
  - RISC-V P Extension (not ratified)
  - CoreV Extension (Custom Vendor extension of Open HW Group, not official)



Packed SIMD vs. Vector

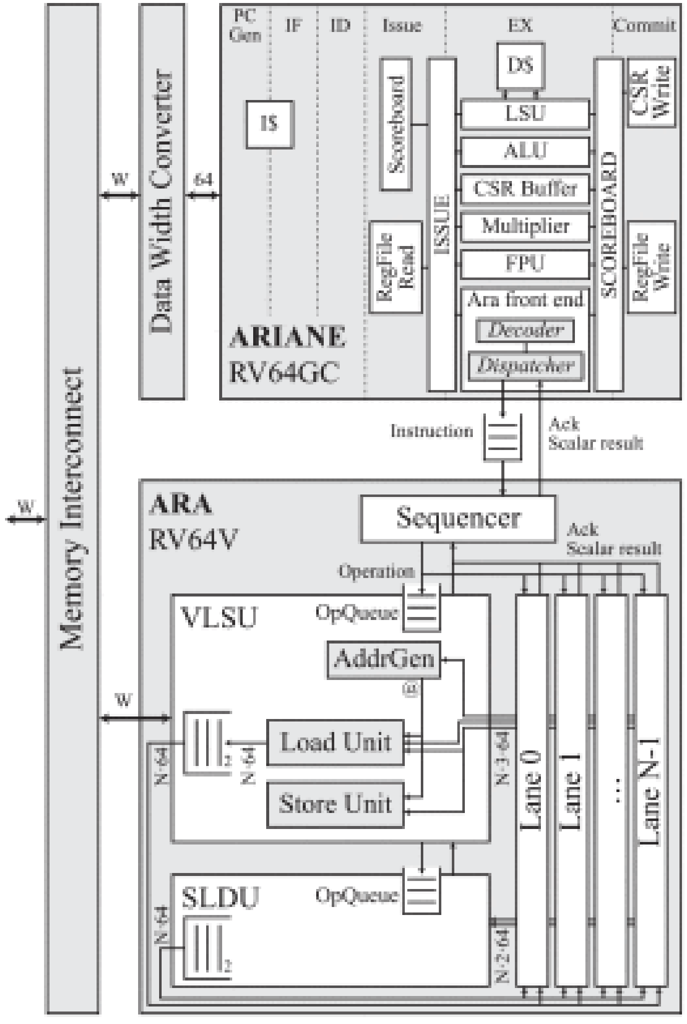
- Pros of Packed SIMD
  - No extra HW Co-processor
  - SIMD unit can share resources in pipeline (ALU and SIMD ALU)

- Cons of Packed SIMD
  - No configurable vector length
  - Usually no wider load/store unit
  - Limited by scalar register sizes

A look at a real vector unit: ARA

Optional, not relevant for exam

ARA Vector Unit



(a)

Vector Unit for the Ariane (now CVA6)
Open source: https://github.com/pulp-platform/ara

Source: Ara: A 1-GHz+ Scalable and Energy-Efficient RISC-V Vector Processor With Multiprecision Floating-Point Support in 22-nm FD-SOI
https://ieeexplore.ieee.org/abstract/document/8918510

Summary

Conclusion

- Vector Units: Data Level Parallelism
- RISC-V Vector Instruction Set

- Next Session:
    - GPUs
    - Accelerators

Thank you for your attention!

Computer Systems

Heterogene Systeme – GPGPUs, TPUs, NPUs

Daniel Mueller-Gritschneder

27.05.2024

Content

- Motivation: Era of Deep Learning
- GP GPUs
- TPUs / NPUs

Optional, not relevant for exam

Motivation: Era of Deep Learning

Optional, not relevant for exam

Use of Data-level Parallelism (DLP)

ML Plattforms are Heterogeneous

- Large computing continuum with possibly connectivity:

Cloud

Datacenter:
Multi-Servers
with Multi-GPUs

Hundreds of CPUs
Hundreds of GBs of DRAM
Several GPUs with
Tens of GB of DRAM
Several TB of Storage

Desktop/Workstation /Fog:
PC with GPU

2-128 CPUs
Tens of GBs of DRAM
1-2 GPUs with Tens of GB of DRAM
A few TB of Storage

Edge/Mobile:
Mobile Phone
Raspberry PI
Embedded GPU
Specialized SoCs

1-4 CPUs
1-4 GBs of DRAM
1 GPUs with a few GB of DRAM
Specialized Accelerators
Tens to Hundreds of GB of Storage

MCU

Extreme Edge / TinyML:
MCU
Specialized low-power SoC

1 CPU
Hundreds of kB to a few MB of embedded SRAM
Low-power Acceleration / Co-proccesors
A few MB of Storage, e.g. embedded Flash

Cloud ML

Desktop ML

Embedded Machine Learning / Edge AI

Deep Learning Models are Heterogeneous

- **In type**: Deep Neural Networks, Convolutional Neural Networks, Transformers, Graph Neural Networks, Recursive Neural Networks
- **In computing demand**: often measured in MAC operations
- **In size**: often measured in number of parameters
- Examples:
    - Large Language Models (LLMs) -produces human-like text
        - GPT-4: 170 trillion (10e12) parameters
        - GPT-3: 175 billion (10e9) parameters
    - ResNet18 – 11 million (10e6) parameters – Image classification e.g. for autonomous driving
    - Keyword Spotting (KWS): 16k-300k (10e3) parameters – Detects keyword in an audio stream, e.g. for Audio wakeup (TinyML)

Example: Convolutional Neural Network

- Consists of layers (structure reprented by data flow graph)

$$
\begin{aligned}
\mathbf{A}^1 &= \mathrm{Conv2D}(\mathbf{X}, \mathbf{W}^1, \mathbf{b}^1, \sigma_s^1, \sigma_r^1, \delta_s^1, \delta_r^1, P^1, \mathrm{ReLu}) \\
\mathbf{A}^2 &= \mathrm{maxpool}(\mathbf{A}^1 \pi_r^2, \pi_s^2) \\
\mathbf{A}^3 &= \mathrm{Conv2D}(\mathbf{A}^2, \mathbf{W}^3, \mathbf{b}^3, \sigma_s^3, \sigma_r^3, \delta_s^3, \delta_r^3, P^3, \mathrm{ReLu}) \\
\mathbf{A}^4 &= \mathrm{maxpool}(\mathbf{A}^3, \pi_r^4, \pi_s^4) \\
\mathbf{a}^5 &= \mathrm{flatten}(\mathbf{A}^4) \\
\mathbf{a}^6 &= \mathrm{Dense}(\mathbf{a}^5, \mathbf{W}^6, \mathbf{b}^6, \mathrm{Softmax})
\end{aligned}
$$

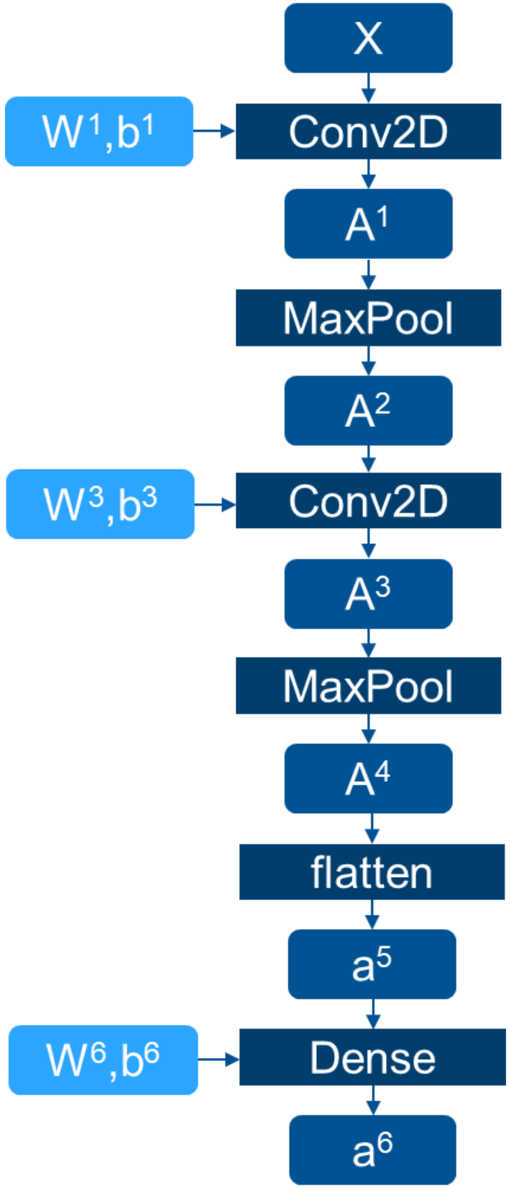


Image to Column (Img2Col) Transformation

- For many targets there exist a very optimized implementation of matrix-matrix-multiply computation e.g. accelerators, for CPUs with some SIMD support, GPUs, but also single-issue CPUs

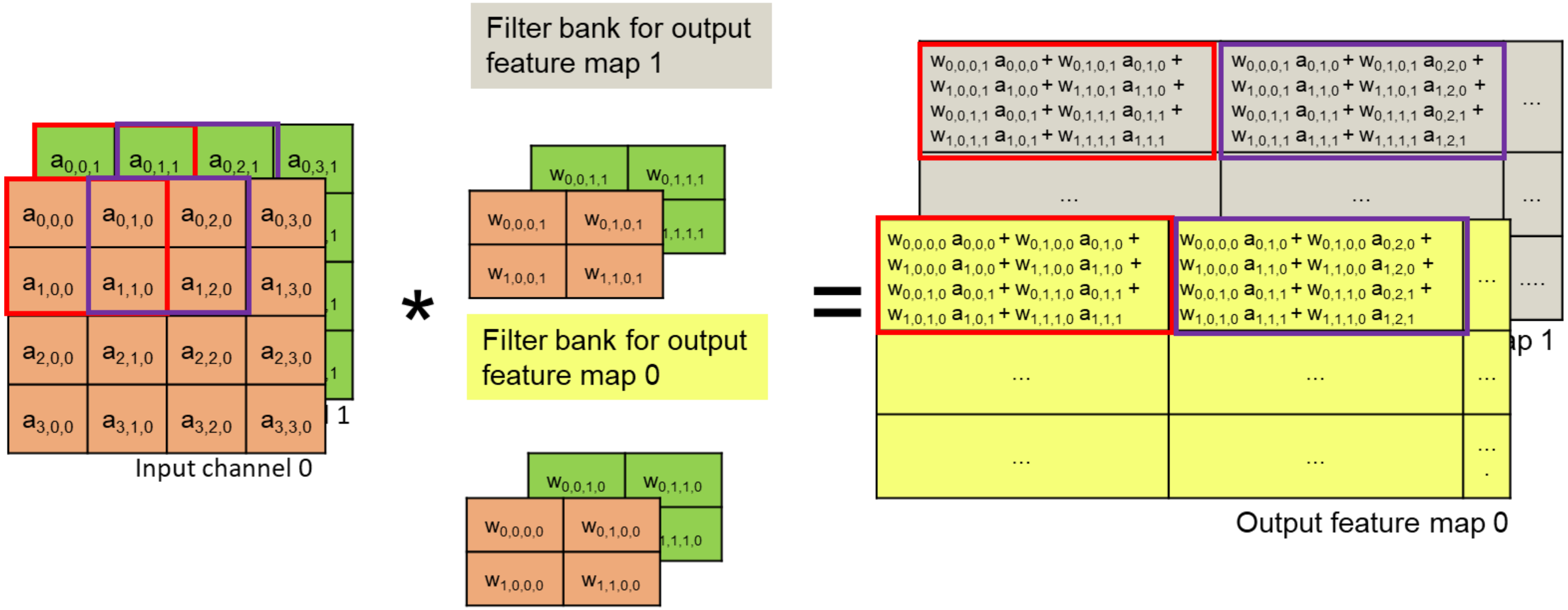
- Img2Col transforms a convolution operation into a matrix-matrix-multiply operation

- Img2Col requires to build up a batch matrix, which is larger than the original activation tensor, because it holds duplicates of some values

➢ Usually Img2Col is not done on the full input activation tensor but inside the convolution loop on some part of the tensor in order to avoid building up the full batch matrix

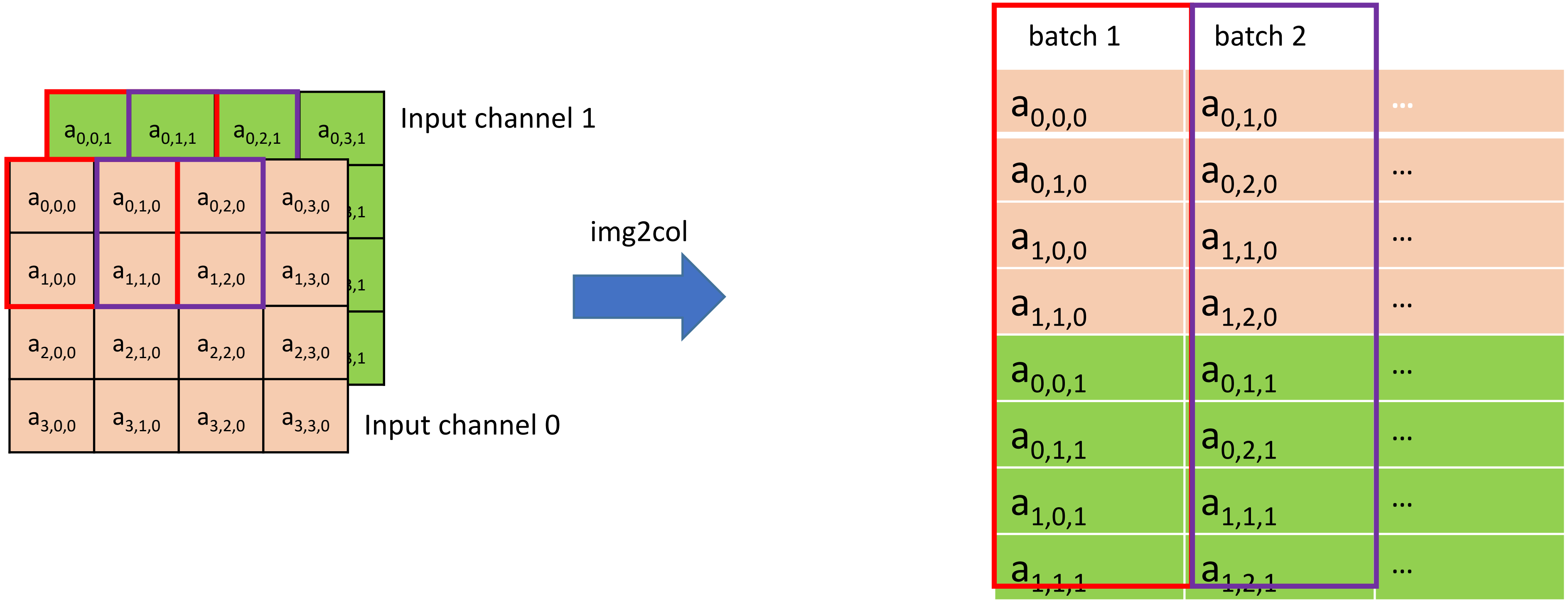
Example for Img2Col (1/5)

- For reference: This is the Standard Convolution


$a_{0,0,1}$ $a_{0,1,1}$ $a_{0,2,1}$ $a_{0,3,1}$
1
$a_{0,0,0}$ $a_{0,1,0}$ $a_{0,2,0}$ $a_{0,3,0}$
$a_{1,0,0}$ $a_{1,1,0}$ $a_{1,2,0}$ $a_{1,3,0}$
$a_{2,0,0}$ $a_{2,1,0}$ $a_{2,2,0}$ $a_{2,3,0}$
$a_{3,0,0}$ $a_{3,1,0}$ $a_{3,2,0}$ $a_{3,3,0}$
Input channel 0
*
Filter bank for output feature map 1
$w_{0,0,1,1}$ $w_{0,1,1,1}$
$w_{0,0,0,1}$ $w_{0,1,0,1}$
$w_{1,0,0,1}$ $w_{1,1,0,1}$
Filter bank for output feature map 0
$w_{0,0,1,0}$ $w_{0,1,1,0}$
$w_{0,0,0,0}$ $w_{0,1,0,0}$
$w_{1,0,0,0}$ $w_{1,1,0,0}$
=
$w_{0,0,0,1}\, a_{0,0,0} + w_{0,1,0,1}\, a_{0,1,0} + w_{1,0,0,1}\, a_{1,0,0} + w_{1,1,0,1}\, a_{1,1,0} + w_{0,0,1,1}\, a_{0,0,1} + w_{0,1,1,1}\, a_{0,1,1} + w_{1,0,1,1}\, a_{1,0,1} + w_{1,1,1,1}\, a_{1,1,1}$
$w_{0,0,0,1}\, a_{0,1,0} + w_{0,1,0,1}\, a_{0,2,0} + w_{1,0,0,1}\, a_{1,1,0} + w_{1,1,0,1}\, a_{1,2,0} + w_{0,0,1,1}\, a_{0,1,1} + w_{0,1,1,1}\, a_{0,2,1} + w_{1,0,1,1}\, a_{1,1,1} + w_{1,1,1,1}\, a_{1,2,1}$
…
ap 1
$w_{0,0,0,0}\, a_{0,0,0} + w_{0,1,0,0}\, a_{0,1,0} + w_{1,0,0,0}\, a_{1,0,0} + w_{1,1,0,0}\, a_{1,1,0} + w_{0,0,1,0}\, a_{0,0,1} + w_{0,1,1,0}\, a_{0,1,1} + w_{1,0,1,0}\, a_{1,0,1} + w_{1,1,1,0}\, a_{1,1,1}$
$w_{0,0,0,0}\, a_{0,1,0} + w_{0,1,0,0}\, a_{0,2,0} + w_{1,0,0,0}\, a_{1,1,0} + w_{1,1,0,0}\, a_{1,2,0} + w_{0,0,1,0}\, a_{0,1,1} + w_{0,1,1,0}\, a_{0,2,1} + w_{1,0,1,0}\, a_{1,1,1} + w_{1,1,1,0}\, a_{1,2,1}$
…
Output feature map 0

Example for Img2Col (2/5)

- Step 1 for Img2Col: Create col-based batch matrix
- Each line holds the activation values under one kernel position for all channels



Input channel 1

$a_{0,0,1}$	$a_{0,1,1}$	$a_{0,2,1}$	$a_{0,3,1}$

Input channel 0

$a_{0,0,0}$	$a_{0,1,0}$	$a_{0,2,0}$	$a_{0,3,0}$
$a_{1,0,0}$	$a_{1,1,0}$	$a_{1,2,0}$	$a_{1,3,0}$
$a_{2,0,0}$	$a_{2,1,0}$	$a_{2,2,0}$	$a_{2,3,0}$
$a_{3,0,0}$	$a_{3,1,0}$	$a_{3,2,0}$	$a_{3,3,0}$

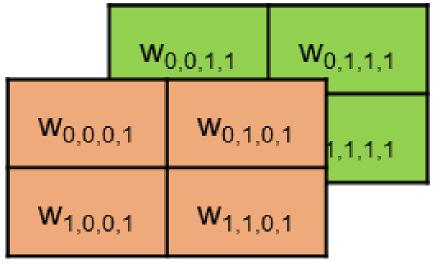
img2col

batch 1	batch 2	
$a_{0,0,0}$	$a_{0,1,0}$	…
$a_{0,1,0}$	$a_{0,2,0}$	…
$a_{1,0,0}$	$a_{1,1,0}$	…
$a_{1,1,0}$	$a_{1,2,0}$	…
$a_{0,0,1}$	$a_{0,1,1}$	…
$a_{0,1,1}$	$a_{0,2,1}$	…
$a_{1,0,1}$	$a_{1,1,1}$	…
$a_{1,1,1}$	$a_{1,2,1}$	…

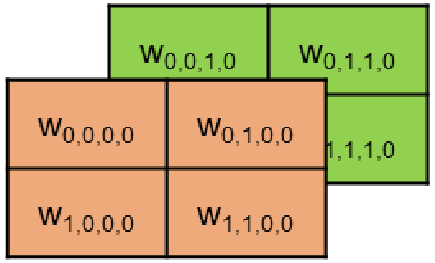
Example for Img2Col (3/5)

- Step 2: Create a row-based filter matrix. (Can be done already offline, is already existing with just storing weight tensor in ROM memory)

Filter bank for output feature map 1 (FM1)



Filter bank for output feature map 0 (FM0)

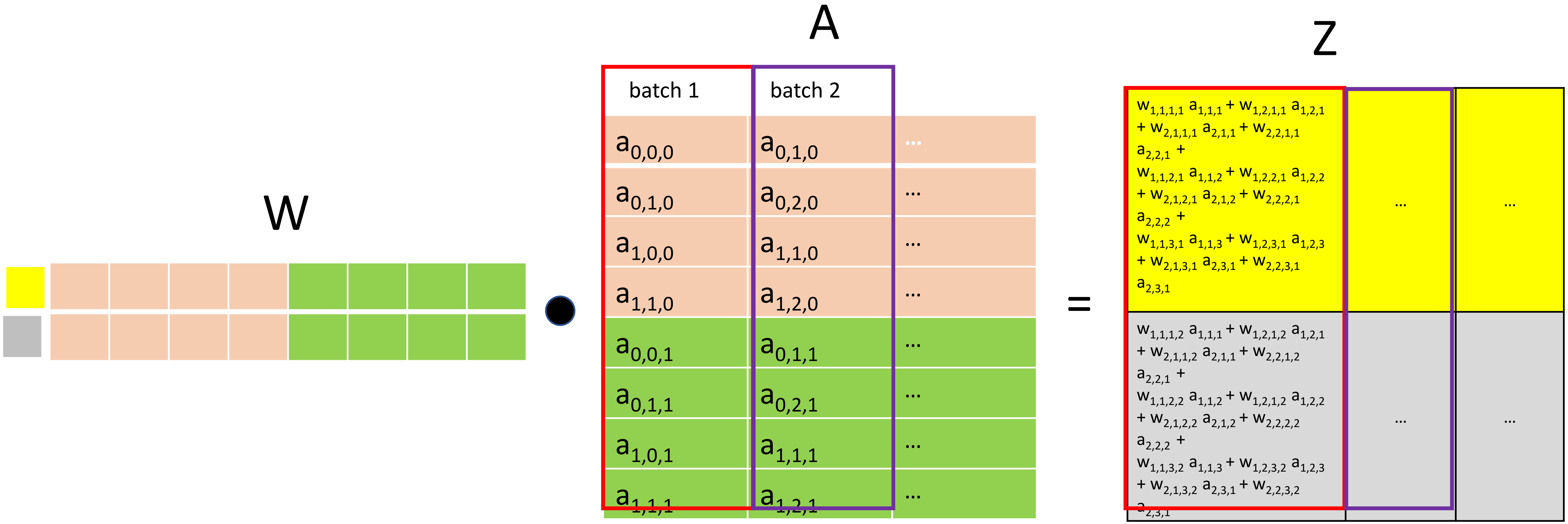


Img2col_weights →

🟨	$w_{0,0,0,0}$	$w_{0,1,0,0}$	$w_{1,0,0,0}$	$w_{1,1,0,0}$	$w_{0,0,1,0}$	$w_{0,1,1,0}$	$w_{1,0,1,0}$	$w_{1,1,1,0}$
⬜	$w_{0,0,0,1}$	$w_{0,1,0,1}$	$w_{1,0,0,1}$	$w_{1,1,0,1}$	$w_{0,0,1,1}$	$w_{0,1,1,1}$	$w_{1,0,1,1}$	$w_{1,1,1,1}$

Example for Img2Col (4/5)

- Step 3: Run a matrix-matrix multiplication with target-specific optimized GEMM kernel

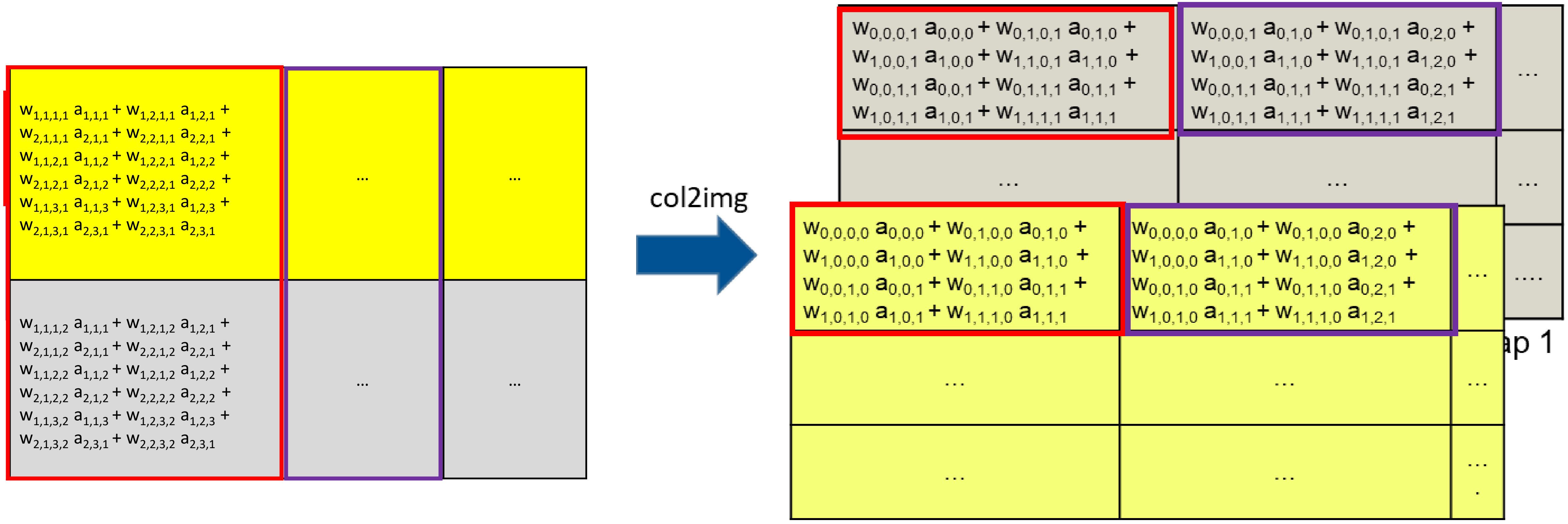


batch 1	batch 2	
$a_{0,0,0}$	$a_{0,1,0}$	…
$a_{0,1,0}$	$a_{0,2,0}$	…
$a_{1,0,0}$	$a_{1,1,0}$	…
$a_{1,1,0}$	$a_{1,2,0}$	…
$a_{0,0,1}$	$a_{0,1,1}$	…
$a_{0,1,1}$	$a_{0,2,1}$	…
$a_{1,0,1}$	$a_{1,1,1}$	…
$a_{1,1,1}$	$a_{1,2,1}$	…

$w_{1,1,1,1}\ a_{1,1,1} + w_{1,2,1,1}\ a_{1,2,1}$ $+ w_{2,1,1,1}\ a_{2,1,1} + w_{2,2,1,1}$ $a_{2,2,1}$ + $w_{1,1,2,1}\ a_{1,1,2} + w_{1,2,2,1}\ a_{1,2,2}$ $+ w_{2,1,2,1}\ a_{2,1,2} + w_{2,2,2,1}$ $a_{2,2,2}$ + $w_{1,1,3,1}\ a_{1,1,3} + w_{1,2,3,1}\ a_{1,2,3}$ $+ w_{2,1,3,1}\ a_{2,3,1} + w_{2,2,3,1}$ $a_{2,3,1}$	…	…
$w_{1,1,1,2}\ a_{1,1,1} + w_{1,2,1,2}\ a_{1,2,1}$ $+ w_{2,1,1,2}\ a_{2,1,1} + w_{2,2,1,2}$ $a_{2,2,1}$ + $w_{1,1,2,2}\ a_{1,1,2} + w_{1,2,1,2}\ a_{1,2,2}$ $+ w_{2,1,2,2}\ a_{2,1,2} + w_{2,2,2,2}$ $a_{2,2,2}$ + $w_{1,1,3,2}\ a_{1,1,3} + w_{1,2,3,2}\ a_{1,2,3}$ $+ w_{2,1,3,2}\ a_{2,3,1} + w_{2,2,3,2}$ $a_{2,3,1}$	…	…

Example for Img2Col (5/5)

- Step 4: Reshape the output to recover the output feature maps using the inverse col2img transformation.



$w_{1,1,1,1}\ a_{1,1,1} + w_{1,2,1,1}\ a_{1,2,1} +$ $w_{2,1,1,1}\ a_{2,1,1} + w_{2,2,1,1}\ a_{2,2,1} +$ $w_{1,1,2,1}\ a_{1,1,2} + w_{1,2,2,1}\ a_{1,2,2} +$ $w_{2,1,2,1}\ a_{2,1,2} + w_{2,2,2,1}\ a_{2,2,2} +$ $w_{1,1,3,1}\ a_{1,1,3} + w_{1,2,3,1}\ a_{1,2,3} +$ $w_{2,1,3,1}\ a_{2,3,1} + w_{2,2,3,1}\ a_{2,3,1}$	…	…
$w_{1,1,1,2}\ a_{1,1,1} + w_{1,2,1,2}\ a_{1,2,1} +$ $w_{2,1,1,2}\ a_{2,1,1} + w_{2,2,1,2}\ a_{2,2,1} +$ $w_{1,1,2,2}\ a_{1,1,2} + w_{1,2,1,2}\ a_{1,2,2} +$ $w_{2,1,2,2}\ a_{2,1,2} + w_{2,2,2,2}\ a_{2,2,2} +$ $w_{1,1,3,2}\ a_{1,1,3} + w_{1,2,3,2}\ a_{1,2,3} +$ $w_{2,1,3,2}\ a_{2,3,1} + w_{2,2,3,2}\ a_{2,3,1}$	…	…

$w_{0,0,0,1}\ a_{0,0,0} + w_{0,1,0,1}\ a_{0,1,0} +$ $w_{1,0,0,1}\ a_{1,0,0} + w_{1,1,0,1}\ a_{1,1,0} +$ $w_{0,0,1,1}\ a_{0,0,1} + w_{0,1,1,1}\ a_{0,1,1} +$ $w_{1,0,1,1}\ a_{1,0,1} + w_{1,1,1,1}\ a_{1,1,1}$	$w_{0,0,0,1}\ a_{0,1,0} + w_{0,1,0,1}\ a_{0,2,0} +$ $w_{1,0,0,1}\ a_{1,1,0} + w_{1,1,0,1}\ a_{1,2,0} +$ $w_{0,0,1,1}\ a_{0,1,1} + w_{0,1,1,1}\ a_{0,2,1} +$ $w_{1,0,1,1}\ a_{1,1,1} + w_{1,1,1,1}\ a_{1,2,1}$	…
…	…	…
		….

$w_{0,0,0,0}\ a_{0,0,0} + w_{0,1,0,0}\ a_{0,1,0} +$ $w_{1,0,0,0}\ a_{1,0,0} + w_{1,1,0,0}\ a_{1,1,0} +$ $w_{0,0,1,0}\ a_{0,0,1} + w_{0,1,1,0}\ a_{0,1,1} +$ $w_{1,0,1,0}\ a_{1,0,1} + w_{1,1,1,0}\ a_{1,1,1}$	$w_{0,0,0,0}\ a_{0,1,0} + w_{0,1,0,0}\ a_{0,2,0} +$ $w_{1,0,0,0}\ a_{1,1,0} + w_{1,1,0,0}\ a_{1,2,0} +$ $w_{0,0,1,0}\ a_{0,1,1} + w_{0,1,1,0}\ a_{0,2,1} +$ $w_{1,0,1,0}\ a_{1,1,1} + w_{1,1,1,0}\ a_{1,2,1}$	…
…	…	…
…	…	… .

Output feature map 0

GEMM Algorithm

- Basic linear algebra algorithm for matrix-matrix-multiply

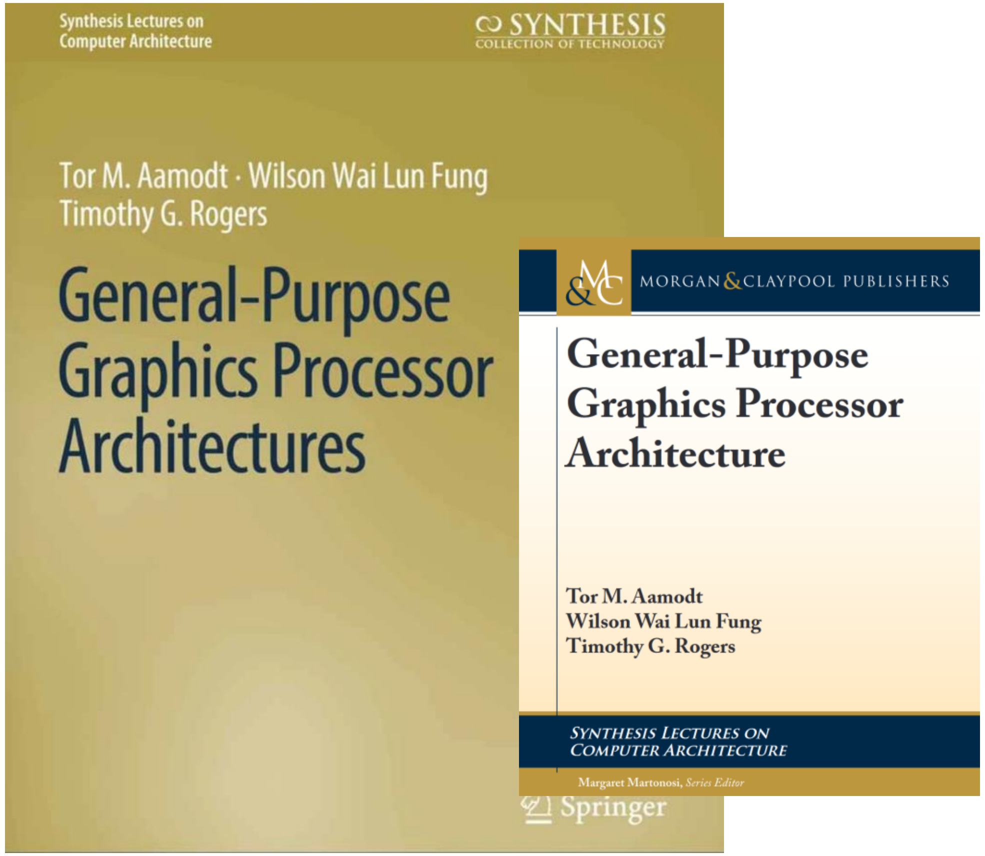
- Optimized versions exist for many hardware platforms e.g.
  - Considering block-wise computation depending on cache sizes
  - Exploiting **data-level parallelism (DLP)**

•GEMM is seen as „*at the heart of deep learning*“ especially when acceleration is considered.

Further reading:
https://petewarden.com/2015/04/20/why-gemm-is-at-the-heart-of-deep-learning/

General-Purpose Graphics Processor Units (GPGPUs)

Source



Synthesis Lectures on Computer Architecture

SYNTHESIS COLLECTION OF TECHNOLOGY

Tor M. Aamodt · Wilson Wai Lun Fung
Timothy G. Rogers

General-Purpose Graphics Processor Architectures

Springer

MORGAN & CLAYPOOL PUBLISHERS

General-Purpose Graphics Processor Architecture

Tor M. Aamodt
Wilson Wai Lun Fung
Timothy G. Rogers

SYNTHESIS LECTURES ON COMPUTER ARCHITECTURE

Margaret Martonosi, *Series Editor*

Inspired by:

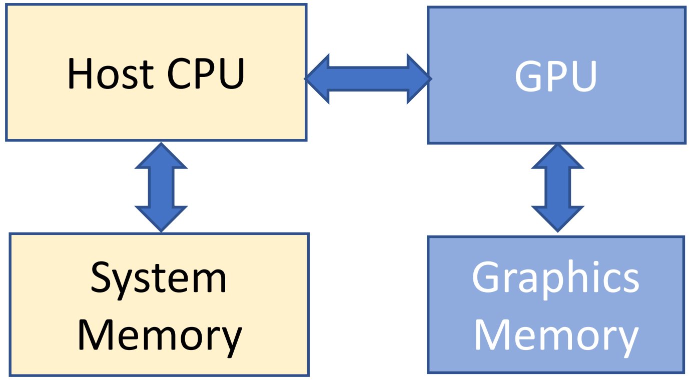
- Book: Aamodt, Fung & Rogers – Generap-Purpose Graphics Processor Architectures
- Book: Hennesy &Patterson: Computer Architecture – A Qualitative Approach
- CA Course: Sophia Shao, UC Berkeley

GPUs

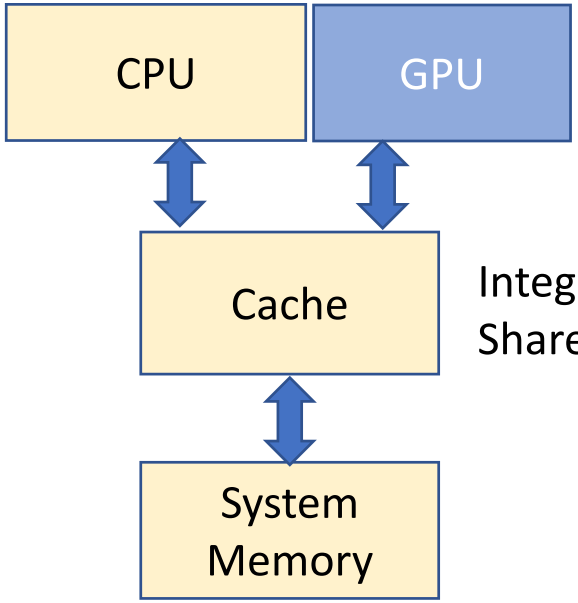
- GPUs were initially introduced for rendering in real time especially for video games.
- Nowadays GPUs can be found in many devices (Data Centers, PCs, Laptop, Phones, Embedded GPUs...)
- General Purpose (GP-GPU): Programming Language CUDA from NVIDIA allowed to use GPUs for other compute besides rendering (now especially used for ML)

GPU (Discrete vs. Integrated)

- GPUs are combined with a CPU either on a single chip or by inserting an additional card (e.g. via PCIe).
- The CPU is responsible for initiating computation on the GPU and transferring data to and from the GPU. The CPU is often called *“the host”*.



Discrete GPU: Own memory



Integrated GPU:
Shared memory

Basic Programming Model

- CPU (Example Code):

```
void saxpy_serial(int n, float a, float *x, float *y) {
   for (int i = 0; i < n; ++i)
       y[i] = a*x[i] + y[i];
}
```

```
…
saxpy_serial(n, 2.0, x, y); // Invoke serial SAXPY kernel
…
```

Basic Programming Model

- GPU (CUDA):

```
__global__ void saxpy(int n, float a, float *x, float *y)
{
int i = blockIdx.x*blockDim.x + threadIdx.x;
if(i<n)
y[i] = a*x[i] + y[i];
}
```

Compute
Kernel

```
…
float *d_x, *d_y;
int nblocks = (n + 255) / 256;
cudaMalloc( &d_x, n * sizeof(float) );
cudaMalloc( &d_y, n * sizeof(float) );
cudaMemcpy( d_x, h_x, n * sizeof(float), cudaMemcpyHostToDevice );
cudaMemcpy( d_y, h_y, n * sizeof(float), cudaMemcpyHostToDevice );
saxpy<<<nblocks, 256>>>(n, 2.0, d_x, d_y);
cudaMemcpy( h_x, d_x, n * sizeof(float), cudaMemcpyDeviceToHost );
...
```

Setup and call kernel
from CPU program

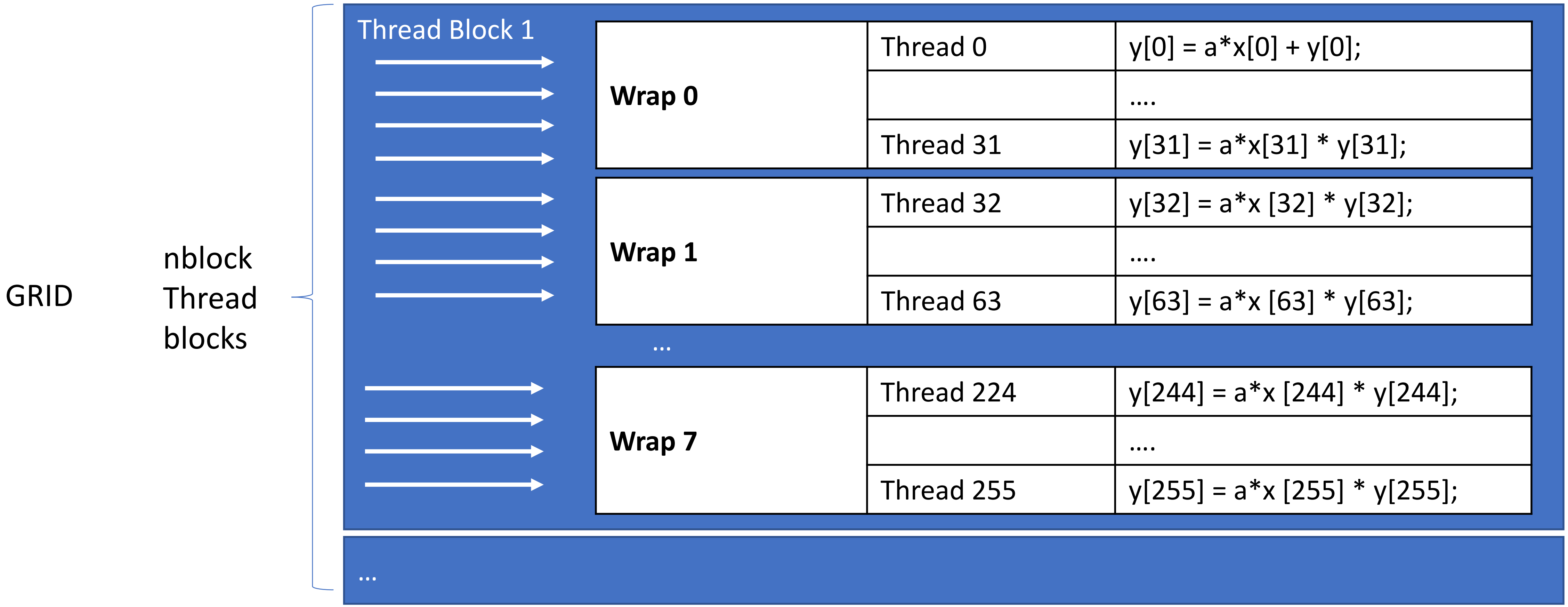
Threads, Warps, Thread block

- The threads that make up a compute kernel are organized into a hierarchy composed of a *grid* of *thread blocks* consisting of *warps*.
- In the CUDA programming model, individual threads execute instructions whose operands are scalar values (e.g., 32-bit floating-point).
- To improve efficiency typical GPU hardware executes groups of threads together in lock-step (SIMD). These groups are called *warps, which* consist of 32 threads
- Warps are grouped into a larger unit called *thread block* by NVIDIA.

Example:

saxpy<<<nblocks, 256>>>(n, 2.0, d_x, d_y);

- Launch a single grid, consisting of nblocks thread blocks
- Each thread block contains 256 threads (8 warps).



GRID

nblock Thread blocks

Thread Block 1

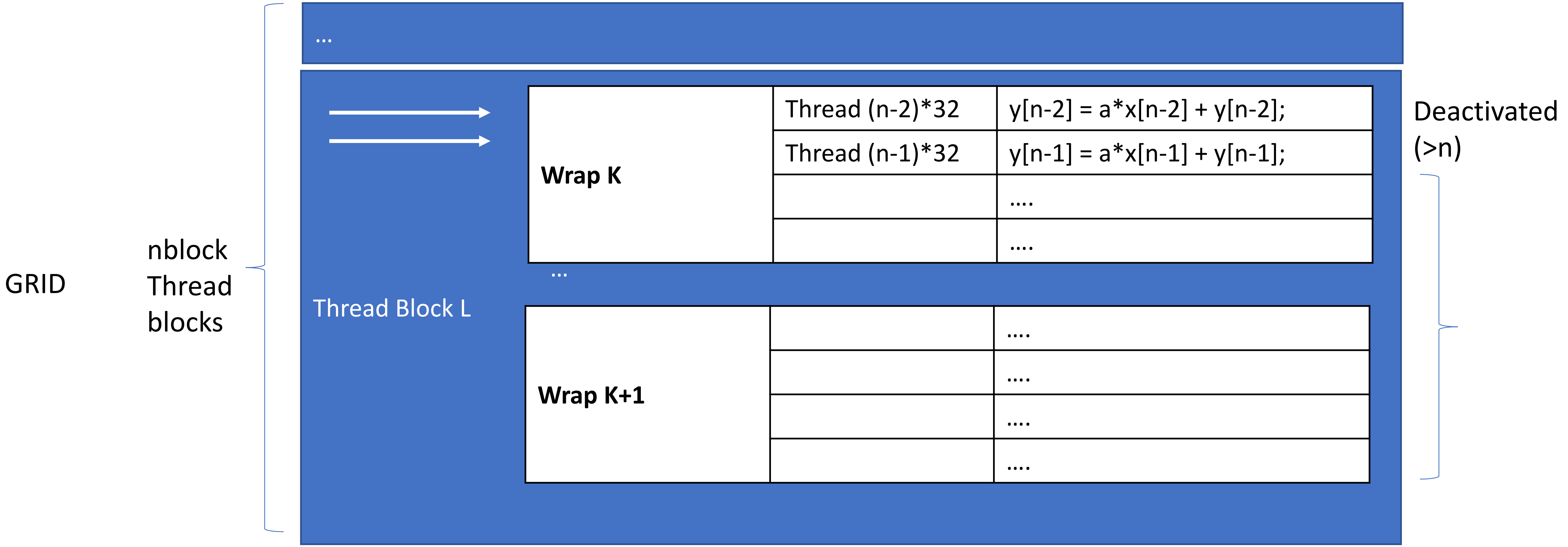
Wrap 0	Thread 0	y[0] = a*x[0] + y[0];
		….
	Thread 31	y[31] = a*x[31] * y[31];
Wrap 1	Thread 32	y[32] = a*x [32] * y[32];
		….
	Thread 63	y[63] = a*x [63] * y[63];
…		
Wrap 7	Thread 224	y[244] = a*x [244] * y[244];
		….
	Thread 255	y[255] = a*x [255] * y[255];

…

Example:

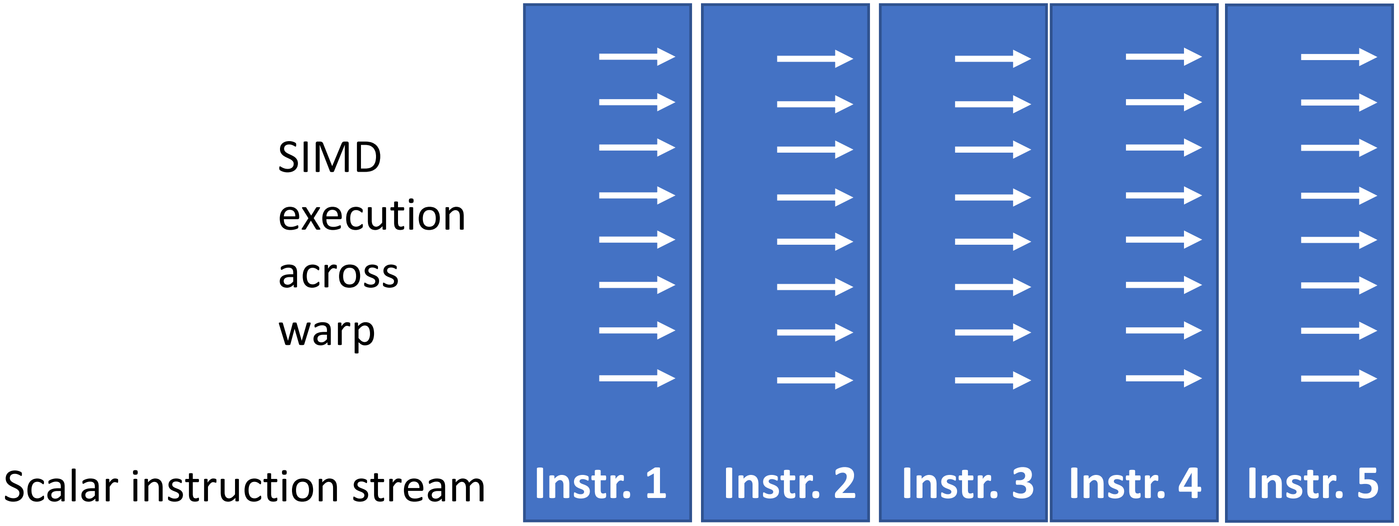
saxpy<<<nblocks, 256>>>(n, 2.0, d_x, d_y);

- Threads with thread_idx.x > n are deactivated



Single Instruction, Multiple Thread (SIMT)

- GPUs uses the **Single Instruction, Multiple Thread (SIMT)** model
- Scalar instruction streams for each CUDA thread are grouped together for SIMD execution on hardware
- Loads and stores are scatter-gather, as threads perform scalar loads and stores



Divergence and Reconvergence of Threads

- Warps execute in lock-step SIMD fashion
- Threads may diverge/reconverge due to control flow

- Simplified illustration (arrows are threads in a thread block):

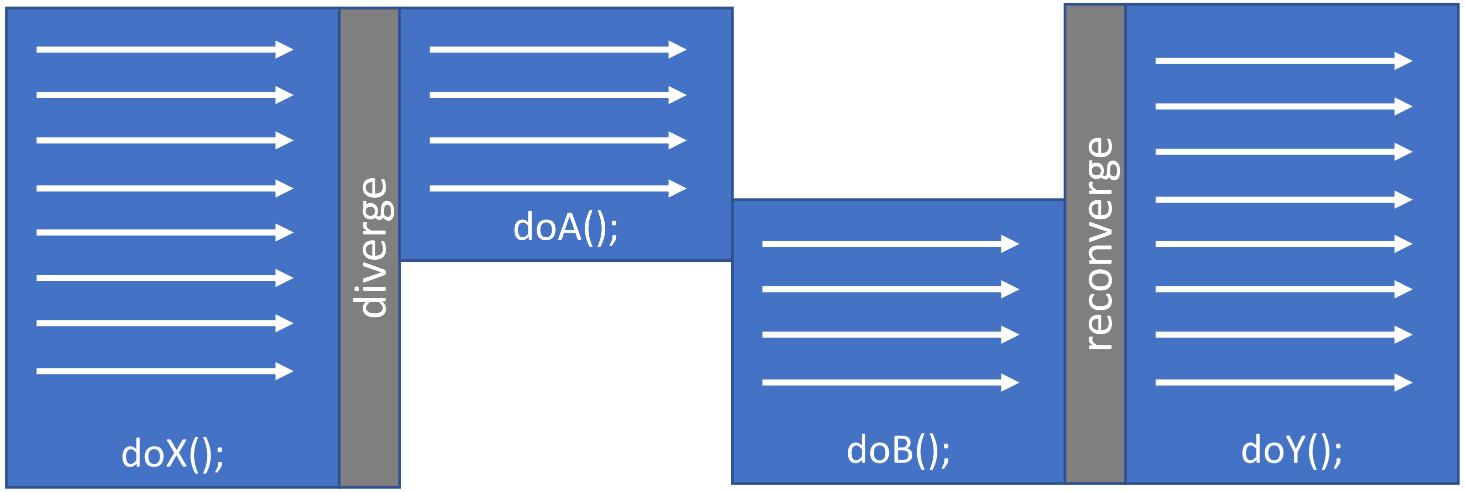
```
doX();
if (threadIdx.x < 4) {
  doA();
} else {
  doB();
}
doY();
```

Mask =
1 1 1 1 1 1 1 1

Mask =
1 1 1 1 0 0 0 0

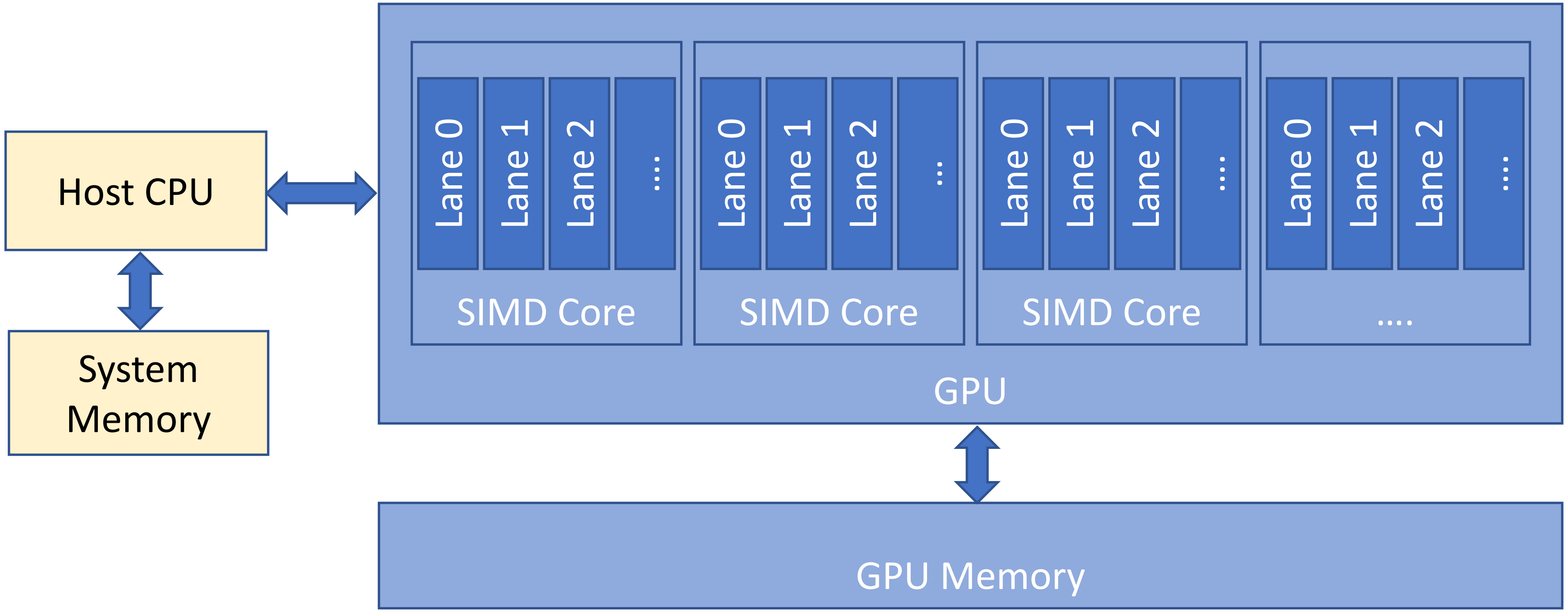
Mask =
0 0 0 0 1 1 1 1

Mask =
1 1 1 1 1 1 1 1



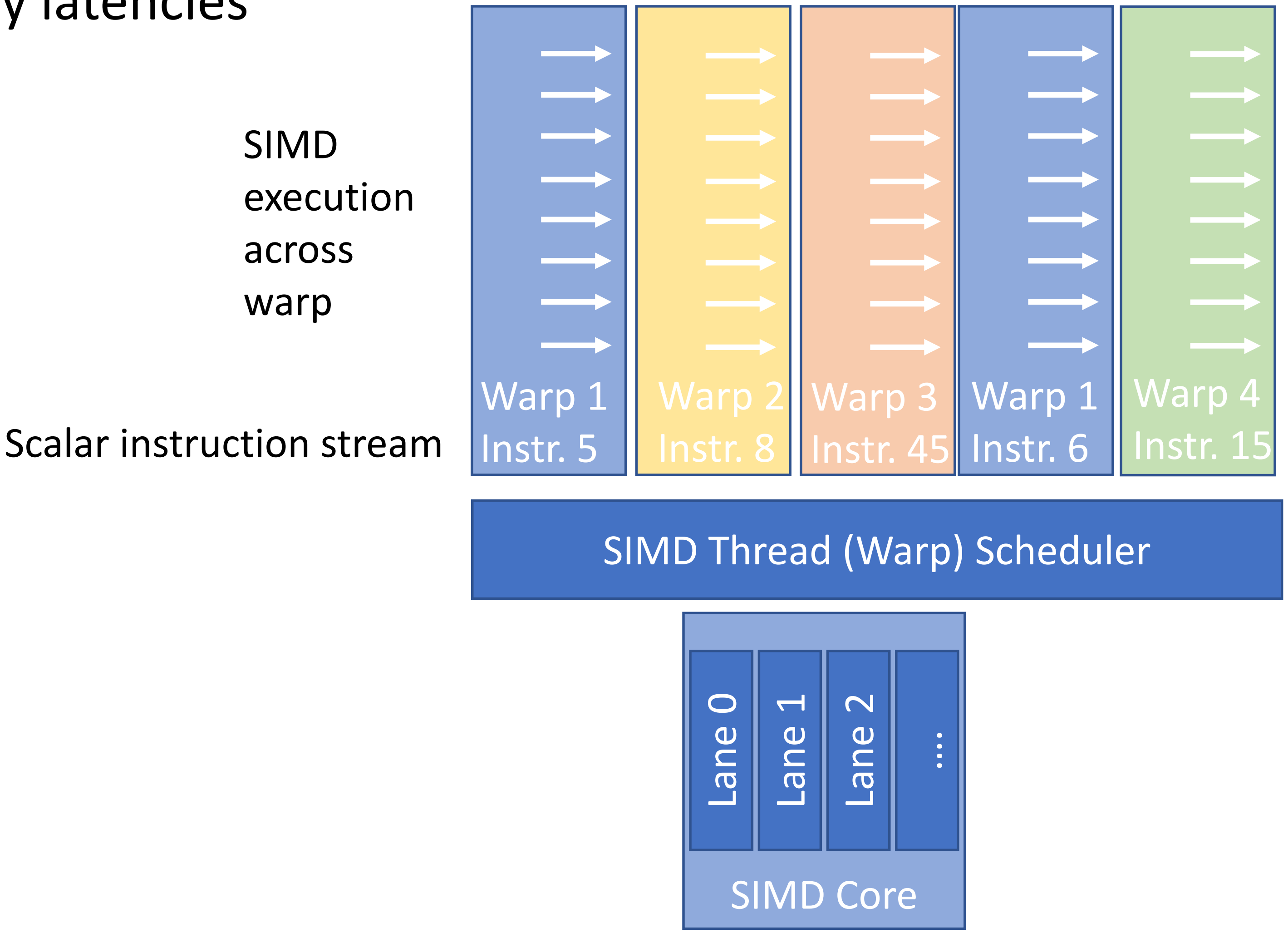
Hardware Execution Model

- GPU is built from multiple parallel cores, each core contains a multithreaded SIMD processor with multiple lanes but with no scalar processor
- CPU sends whole “grid” over to GPU, which distributes thread blocks among cores (each thread block executes on one core)

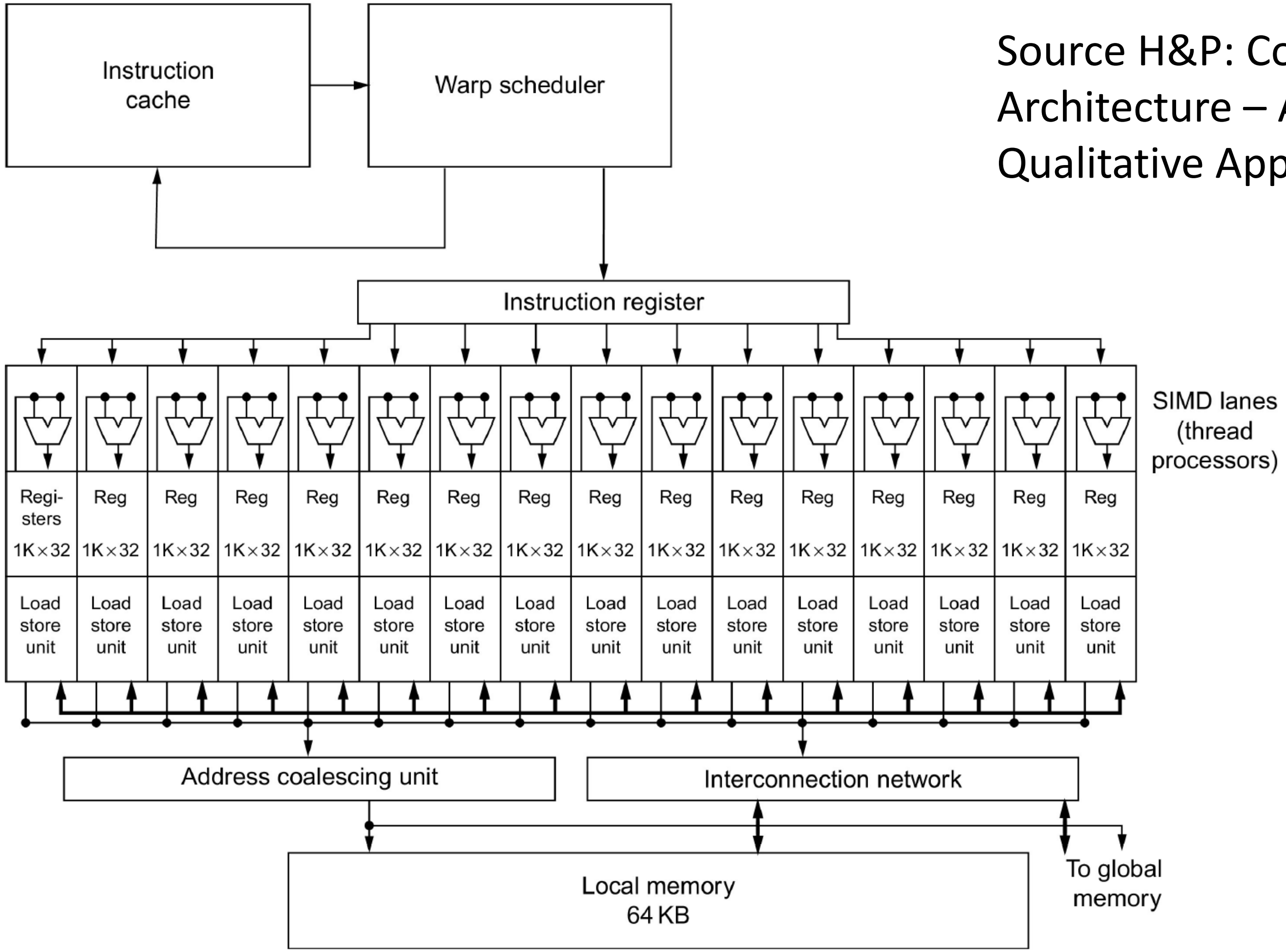


Multithreading on SIMD Processor

- SIMD cores execute instructions of independent warps in multithreaded fashion
- E.g. can hide memory latencies



Multithreaded SIMD Processor



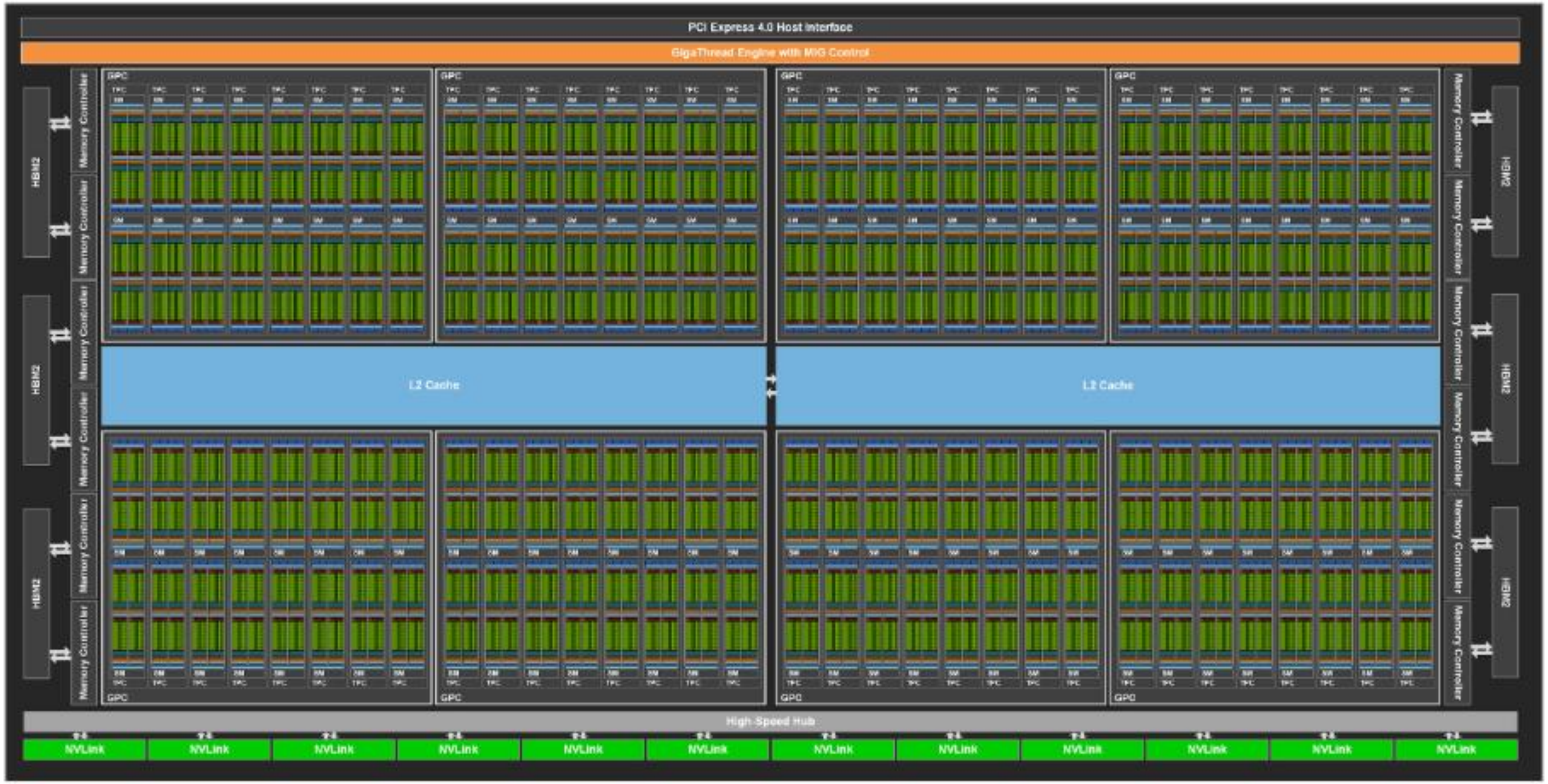
Source H&P: Computer Architecture – A Qualitative Approach

Figure 4.14 Simplified block diagram of a multithreaded SIMD Processor. It has 16 SIMD Lanes. The SIMD Thread Scheduler has, say, 64 independent threads of SIMD instructions that it schedules with a table of 64 program counters (PCs). Note that each lane has 1024 32-bit registers.

Look at a Real GPU: A100

Optional, not relevant for exam

A100 GPU -128 Streaming Multiprocessor



NVIDEA calls
SIMD processors
Streaming Multiprocessors
(SMs)

Source: https://images.nvidia.com/aem-dam/en-zz/Solutions/data-center/nvidia-ampere-architecture-whitepaper.pdf

SM

- “A100 has four Tensor Cores per SM, which together deliver 1024 dense FP16/FP32 FMA operations per clock”
- “432 Third-generation Tensor Cores per GPU” (108 SMs)

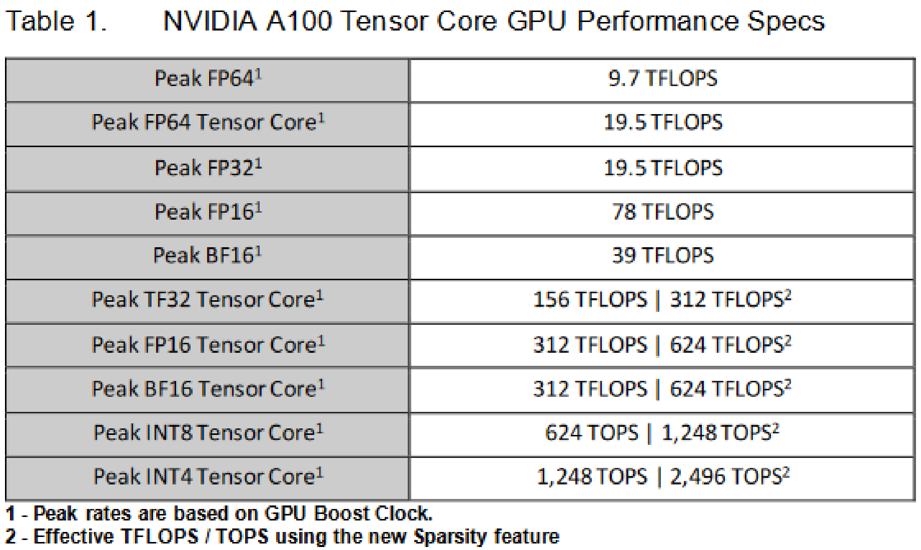
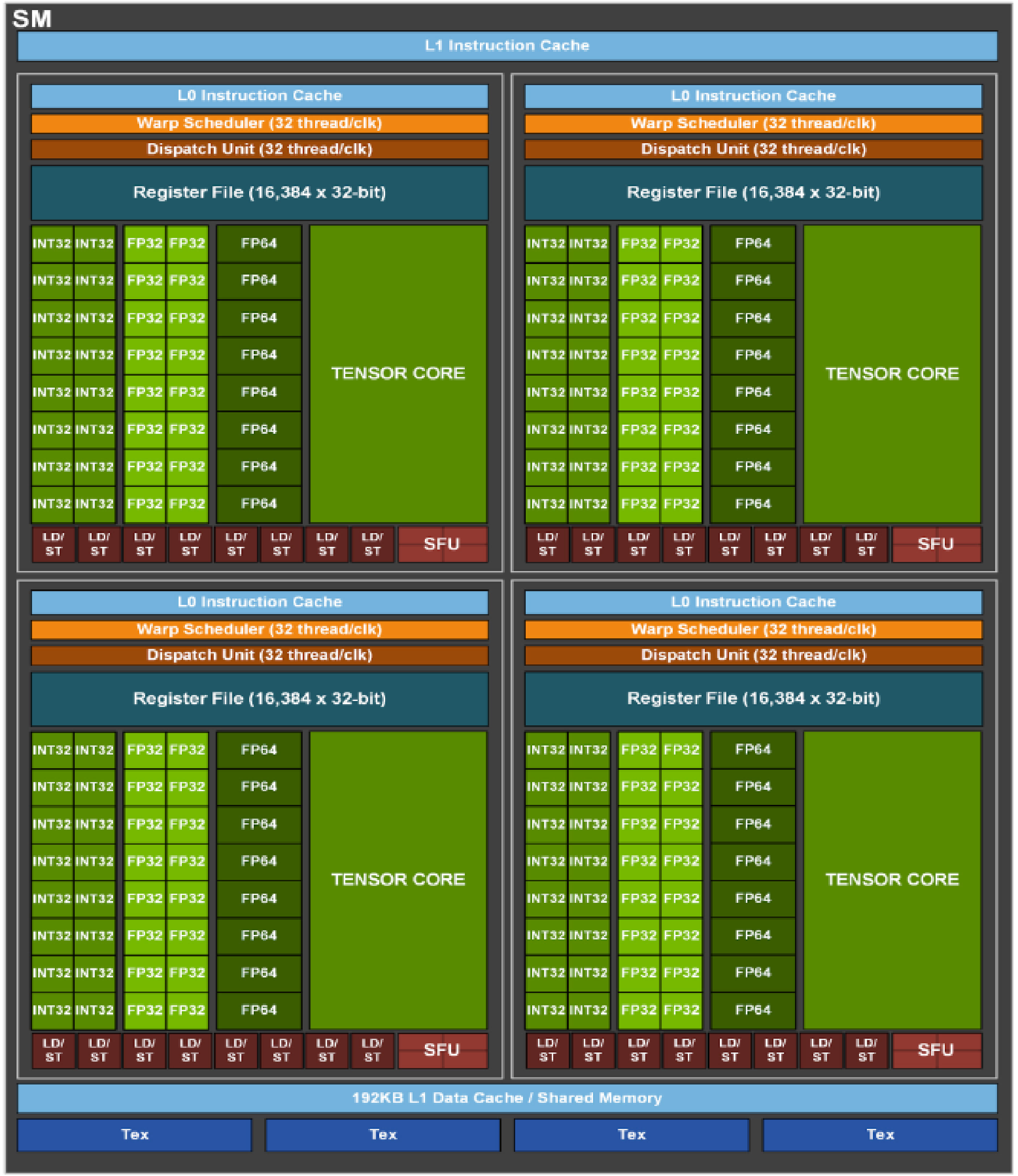


Table 1. NVIDIA A100 Tensor Core GPU Performance Specs

Peak FP64[1]	9.7 TFLOPS
Peak FP64 Tensor Core[1]	19.5 TFLOPS
Peak FP32[1]	19.5 TFLOPS
Peak FP16[1]	78 TFLOPS
Peak BF16[1]	39 TFLOPS
Peak TF32 Tensor Core[1]	156 TFLOPS | 312 TFLOPS[2]
Peak FP16 Tensor Core[1]	312 TFLOPS | 624 TFLOPS[2]
Peak BF16 Tensor Core[1]	312 TFLOPS | 624 TFLOPS[2]
Peak INT8 Tensor Core[1]	624 TOPS | 1,248 TOPS[2]
Peak INT4 Tensor Core[1]	1,248 TOPS | 2,496 TOPS[2]

1 - Peak rates are based on GPU Boost Clock.
2 - Effective TFLOPS / TOPS using the new Sparsity feature


SM
L1 Instruction Cache
L0 Instruction Cache
Warp Scheduler (32 thread/clk)
Dispatch Unit (32 thread/clk)
Register File (16,384 x 32-bit)
INT32
FP32
FP64
TENSOR CORE
LD/ST
SFU
192KB L1 Data Cache / Shared Memory
Tex

Source: https://images.nvidia.com/aem-dam/en-zz/Solutions/data-center/nvidia-ampere-architecture-whitepaper.pdf

Accelerators - Systolic Array

Systolic Array

Concept:

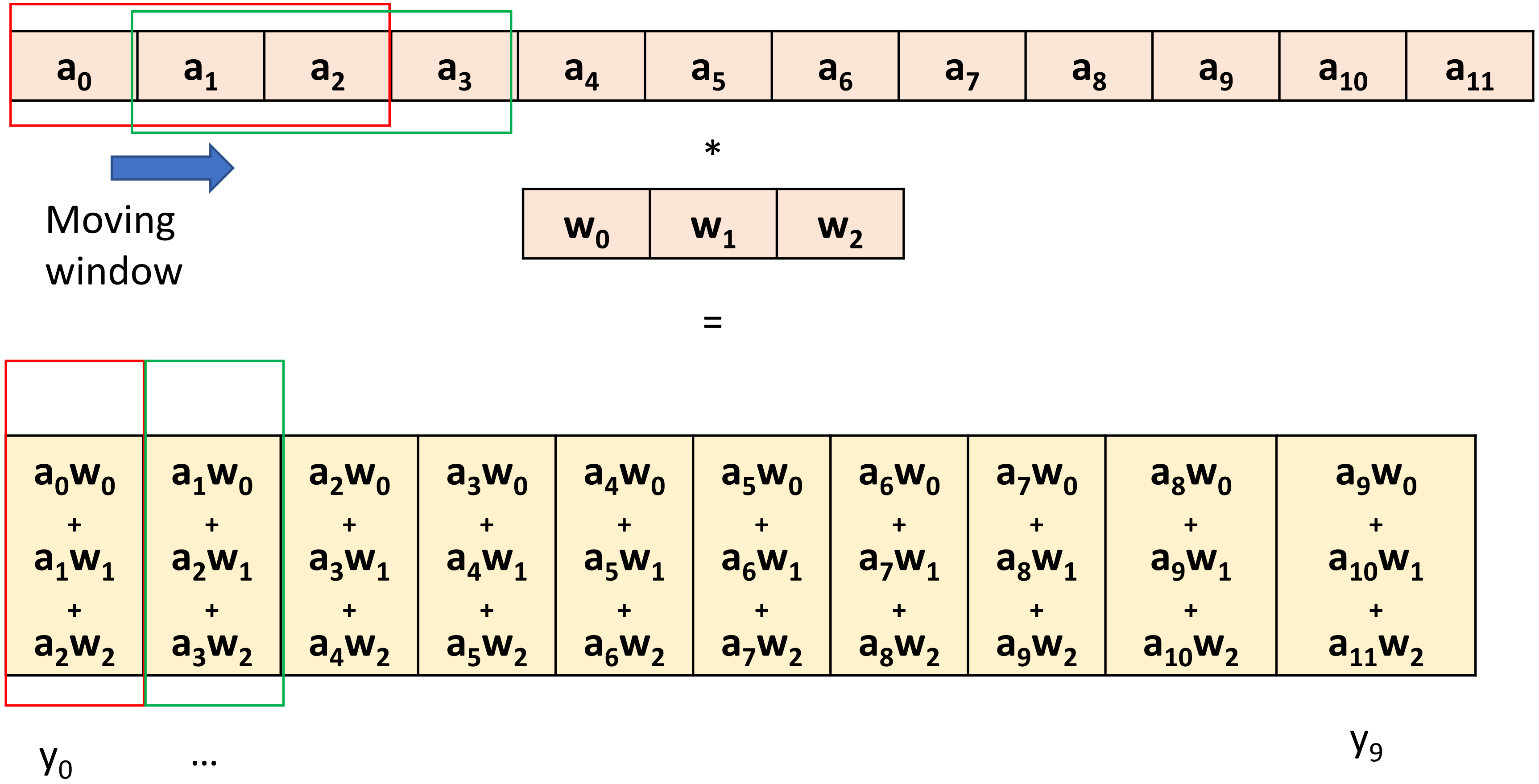
- Functional Units (FUs) are chained to implement a fixed type of computation
- Flow inside systolic array needs to be carefully orchestrated
- Intermediate results are directly moved to next FU

- 2D systolic arrays often used for deep learning for Matrix-matrix multiply (GEMM), called *Tensor Cores*, *GEMM Core, Matrix Multiply Unit*
- Systolic arrays can be designed for many other computations

Example: 1D Convolution

```
void conv1D_12_3(int* x, int* w, int* y) {
for (i=0; i<10;i++) {
  y[i]=0;
  for (j=0;j<3;j++) {
    y[i] += x[i+j] * w[j];
  }
}
}
```

- Simple 1D convolution (A1x12)*(1x3):



a_0	a_1	a_2	a_3	a_4	a_5	a_6	a_7	a_8	a_9	a_{10}	a_{11}

*

w_0	w_1	w_2

=

a_0w_0 + a_1w_1 + a_2w_2	a_1w_0 + a_2w_1 + a_3w_2	a_2w_0 + a_3w_1 + a_4w_2	a_3w_0 + a_4w_1 + a_5w_2	a_4w_0 + a_5w_1 + a_6w_2	a_5w_0 + a_6w_1 + a_7w_2	a_6w_0 + a_7w_1 + a_8w_2	a_7w_0 + a_8w_1 + a_9w_2	a_8w_0 + a_9w_1 + $a_{10}w_2$	a_9w_0 + $a_{10}w_1$ + $a_{11}w_2$

Example: 1D Convolution

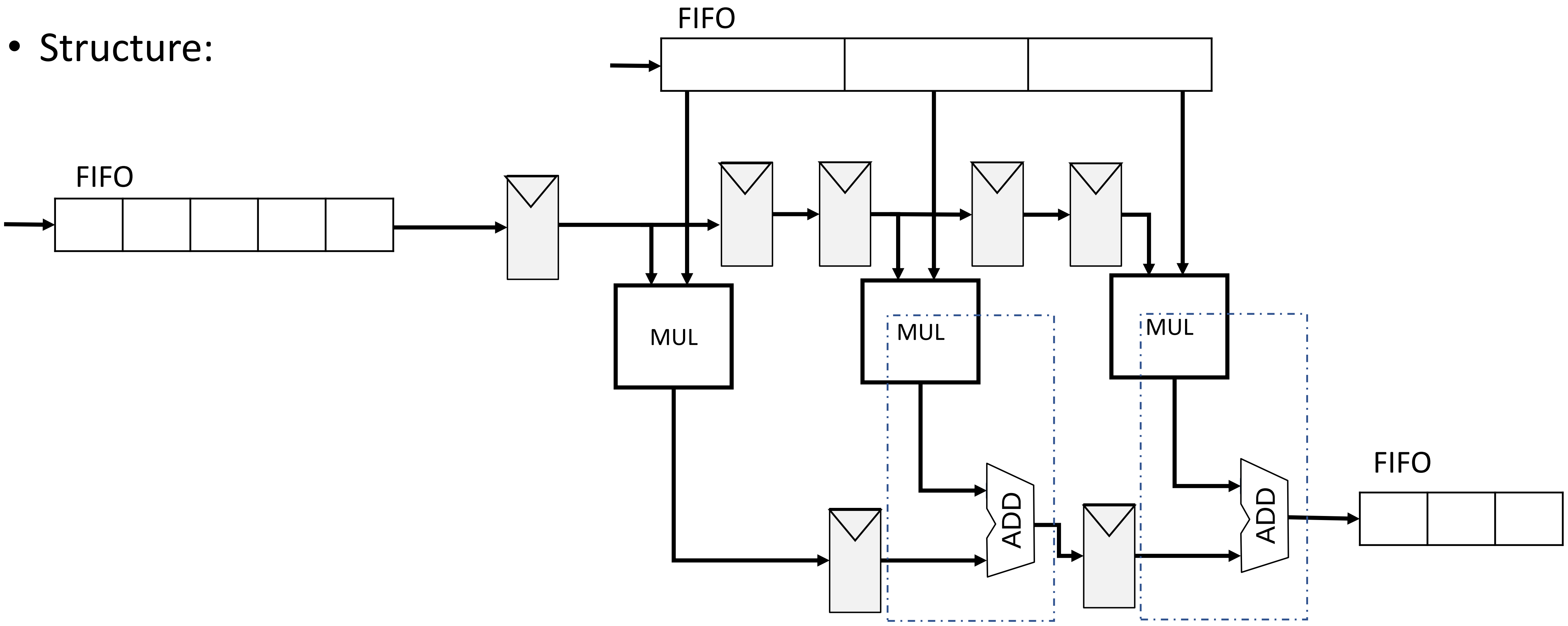
- Code

```
void conv1D_12_3(int* x, int* w, int* y) {
for (i=0; i<10;i++) {
  y[i]=0;
  for (j=0;j<3;j++) {
    y[i] += x[i+j] * w[j];
  }
}
}
```

```
conv1D_12_3:
  LW t1,0(a1)   # w0
  LW t2,4(a1)   # w1
  LW t3,8(a1)   # w2
  LI t4,0
conv1D_12_3_loop:
  LW a4,0(a0)    # x[i+0]
  LW a4,4(a0)    # x[i+1]
  MUL a1,a4,t1  # x[i+0] * w[0]
  MUL a4,a4,t2  # x[i+1] * w[1]
  LW a5,8(a0)    # x[i+2]
  ADD a1,a1,a4
  MUL a5,a5,t3  # x[i+2] * w[2]
  ADD a1,a1,a5
  SW a1,0(a2)  # Store y[i]
  ADDI a0,a0,4
  ADDI a1,a1,4
  ADDI t4,t4,1
  BNE t4,10, conv1D_12_3_loop
RET
```

Example: 1D Convolution - Systolic Array (1D) - Structure

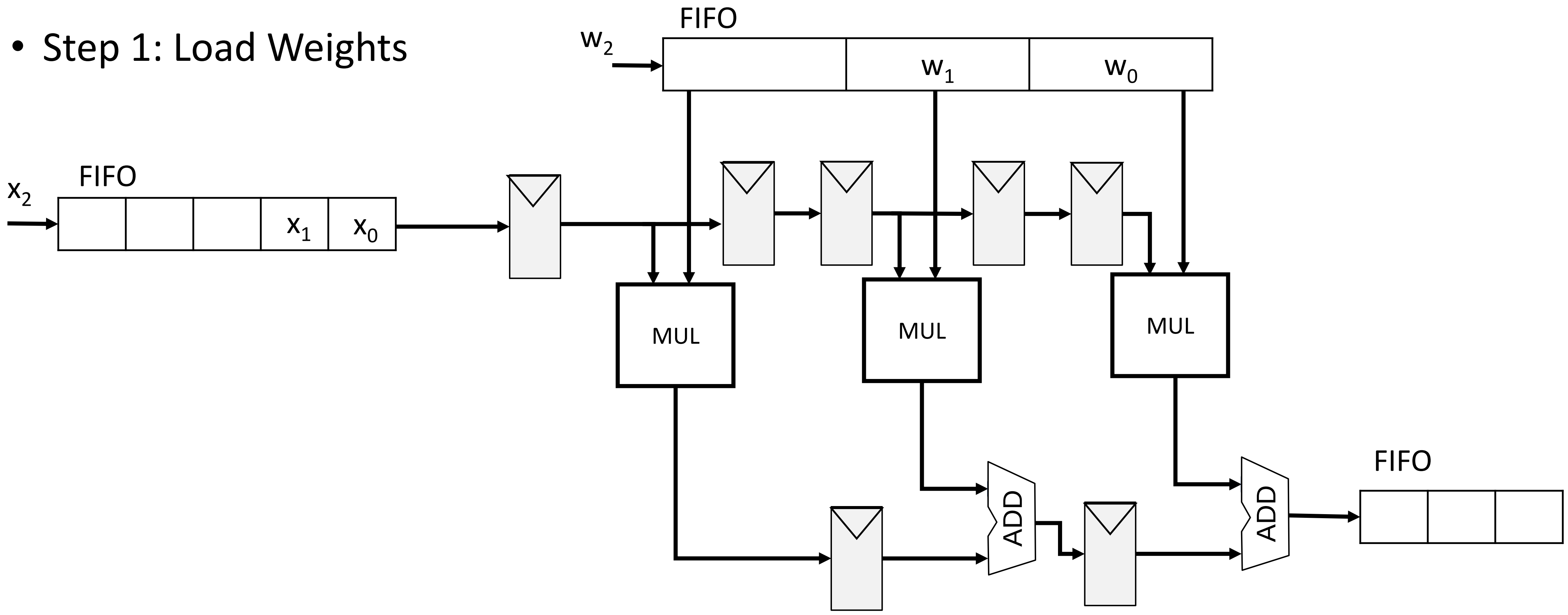
- Structure:



MUL+ ADD FU is called
Multiply-Accumulate (MAC) Unit

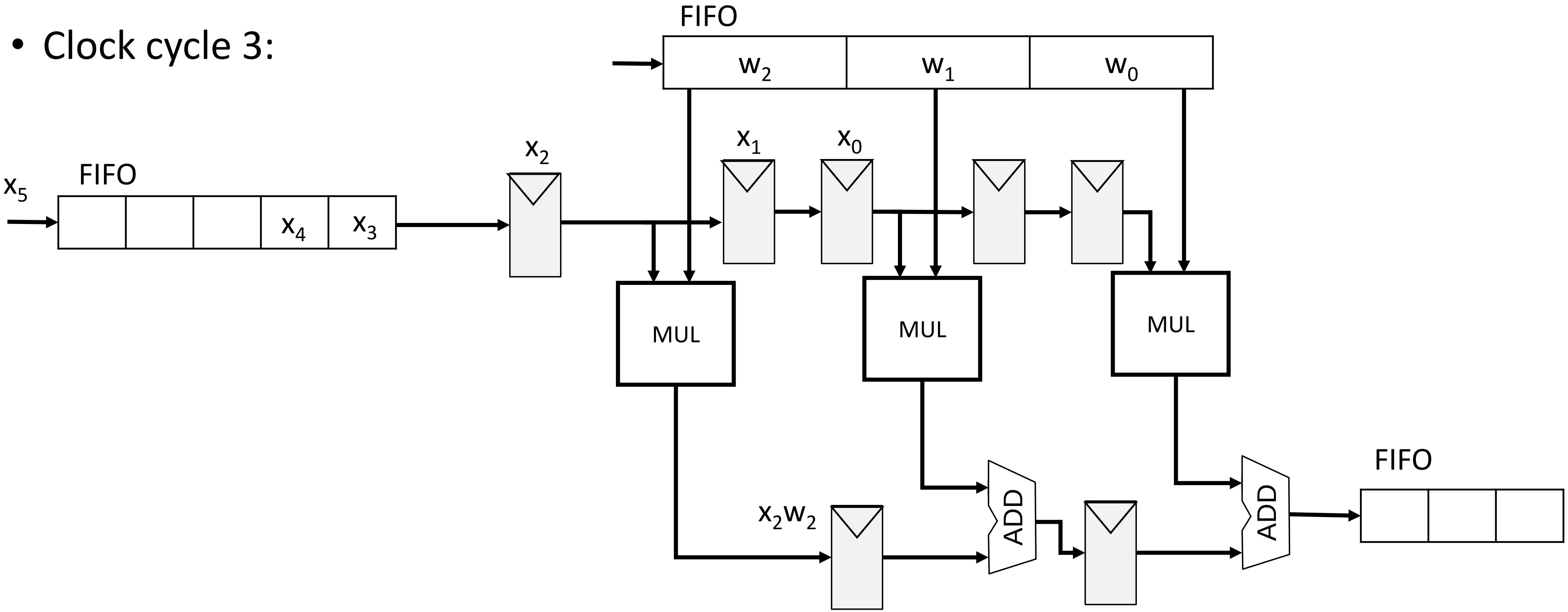
Example: 1D Convolution - Systolic Array (1D) - Structure

- Step 1: Load Weights



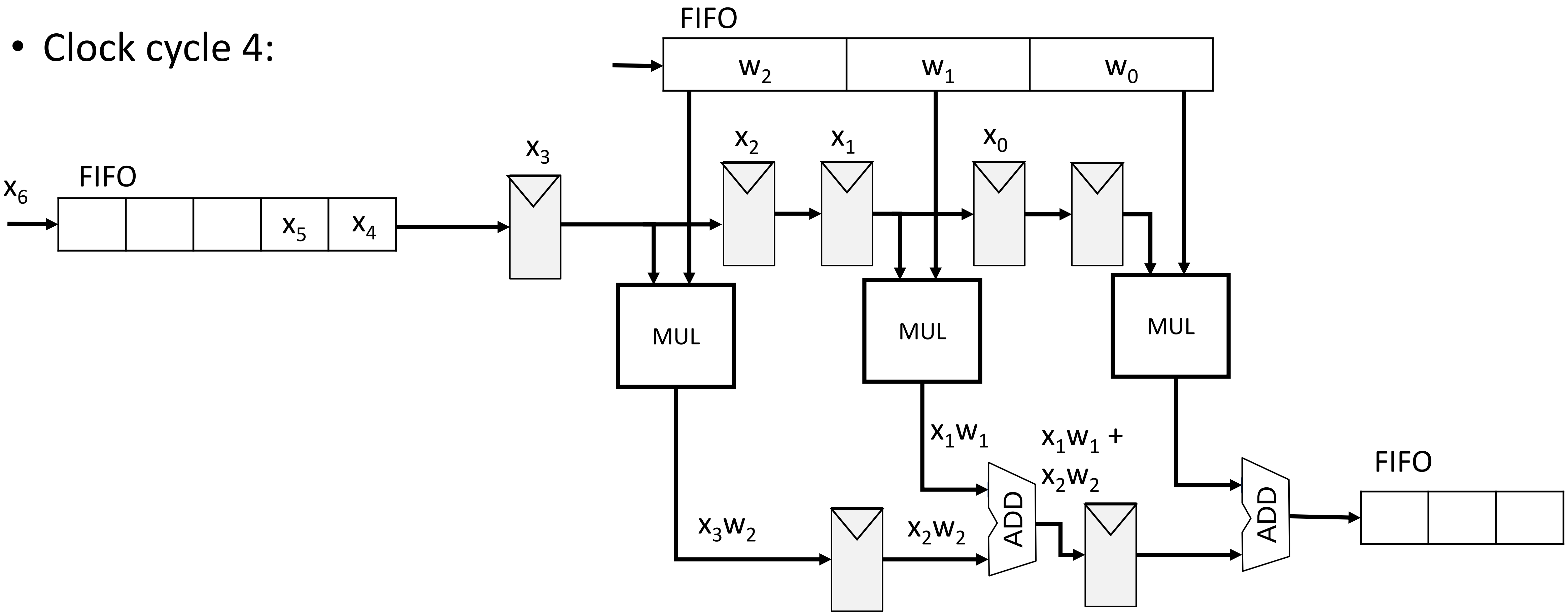
Example: 1D Convolution - Systolic Array (1D) - Structure

- Clock cycle 3:



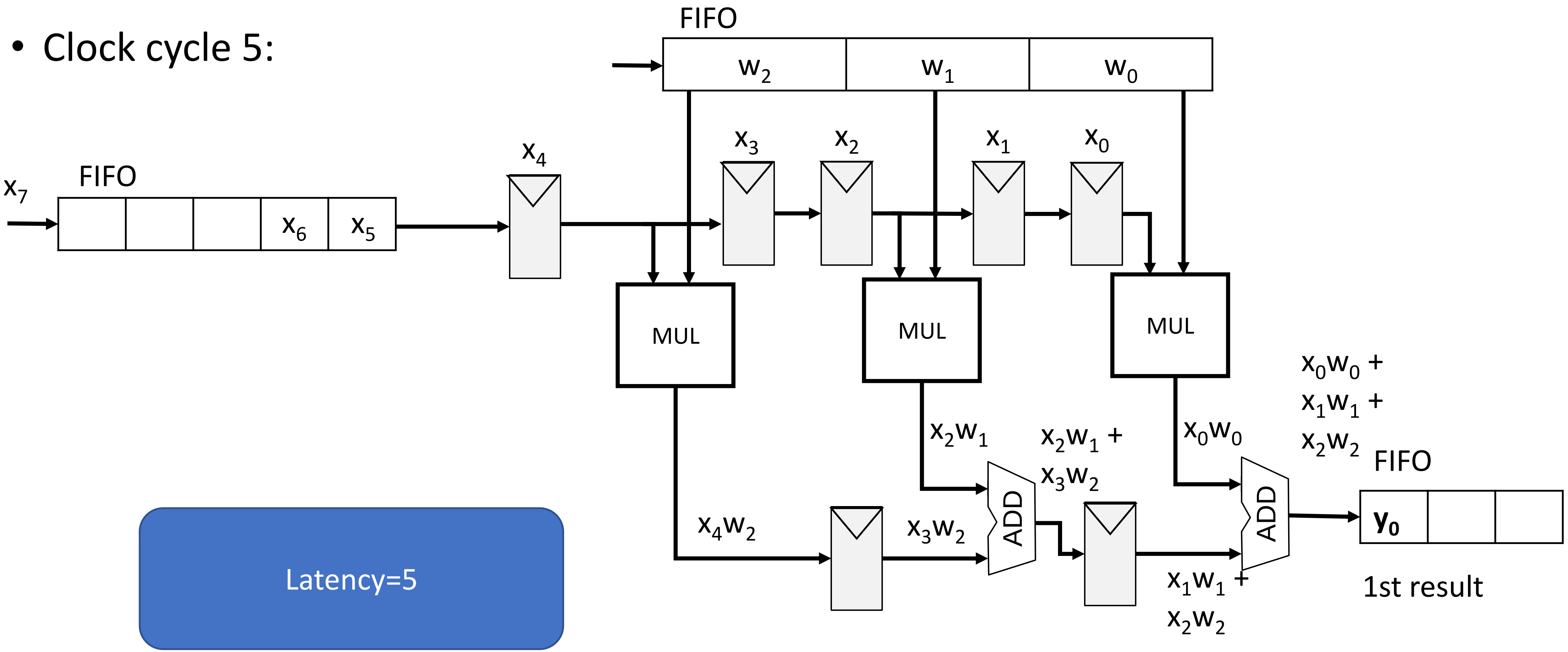
Example: 1D Convolution - Systolic Array (1D) - Structure

- Clock cycle 4:



Example: 1D Convolution - Systolic Array (1D) - Structure

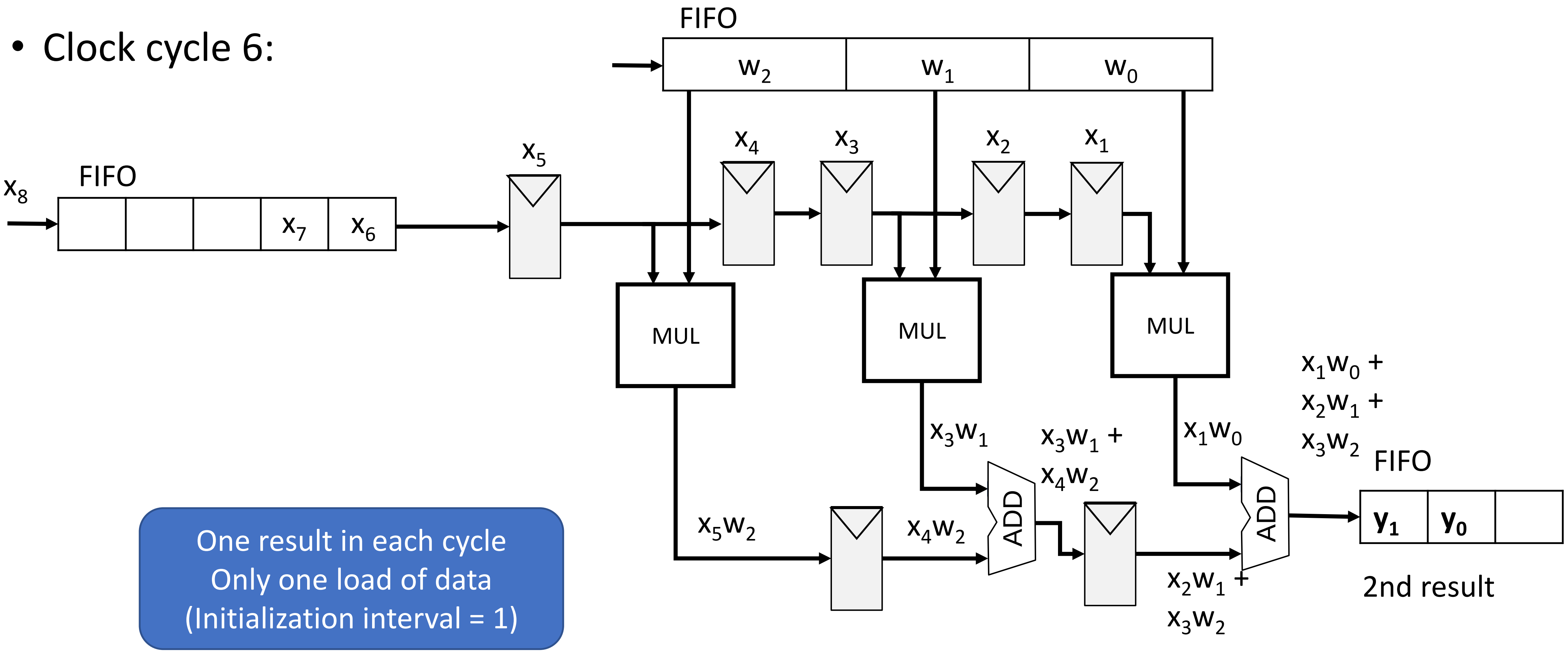
- Clock cycle 5:



Latency=5

Example: 1D Convolution - Systolic Array (1D) - Structure

- Clock cycle 6:



One result in each cycle
Only one load of data
(Initialization interval = 1)

Systolic Arrays Pros-Cons

- Advantages:
  - Move intermediate results between FUs to reduce memory access
  - Balance between computation and memory bandwidth
  - Simple design to exploit **data-level parallelism** (DLP)

  - Different systolic arrays can be combined for multi-stage computations

- Disadvantage
  - Specialized: computation needs to fit FU arrangement

A look at Real ML Accelerators

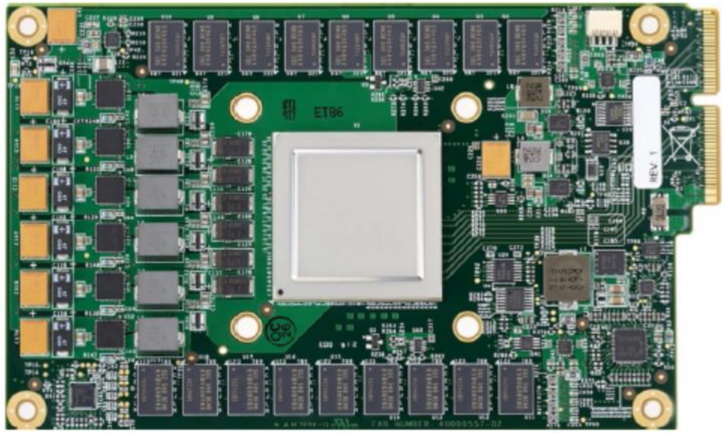
Optional, not relevant for exam

Google Tensor Processing Unit (TPU)

VTA Neural Processing Unit (NPU)

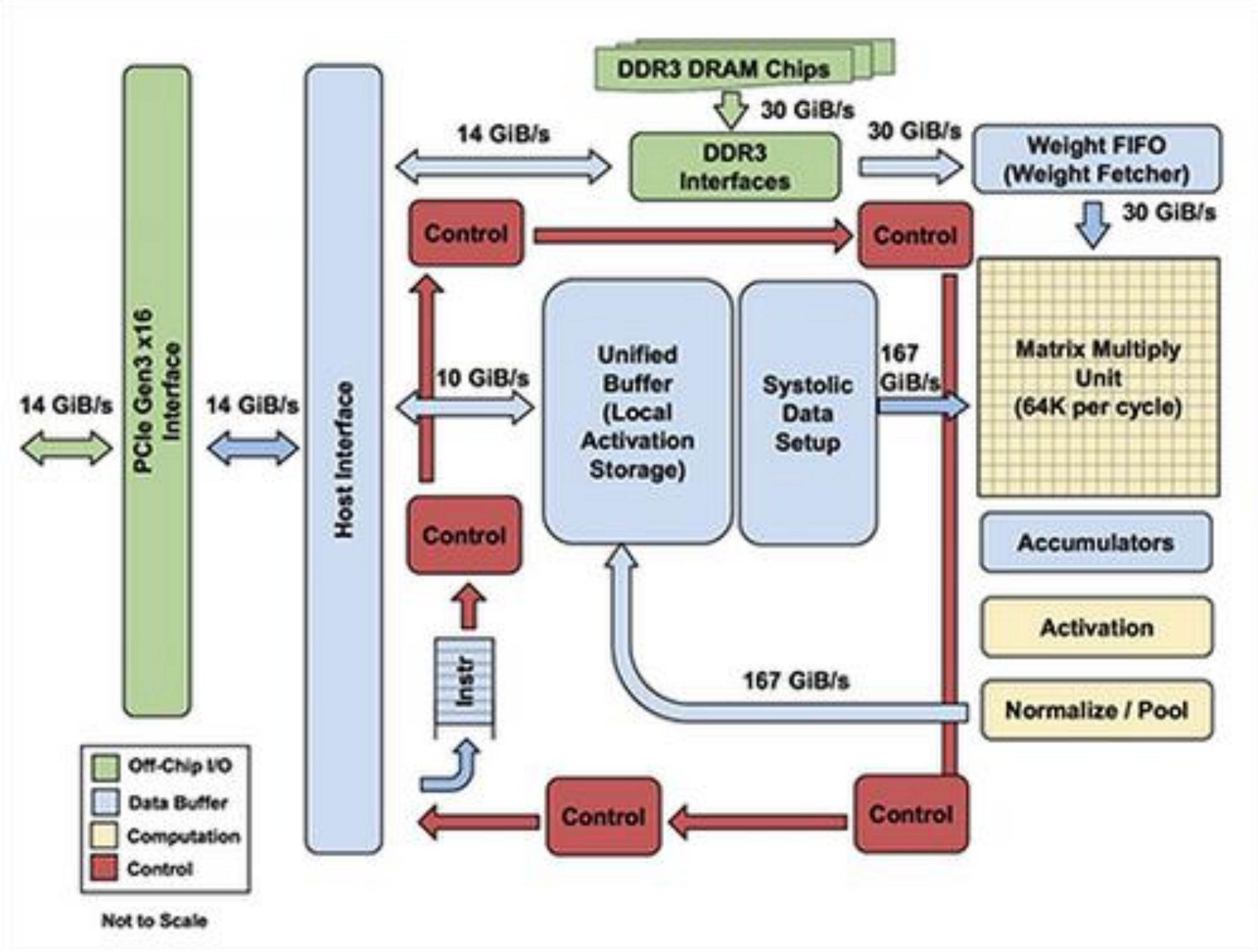
TPU V1: Tensor Processing Unit (2017)

- Application-specific Integrated Circuit (ASIC) – Chip from Google
- Specialized to accelerate Deep Neural Network (DNN) computations
- PCB board with PCIe Interface to Host processor



Source: https://cloud.google.com/blog/products/ai-machine-learning/an-in-depth-look-at-googles-first-tensor-processing-unit-tpu?hl=en

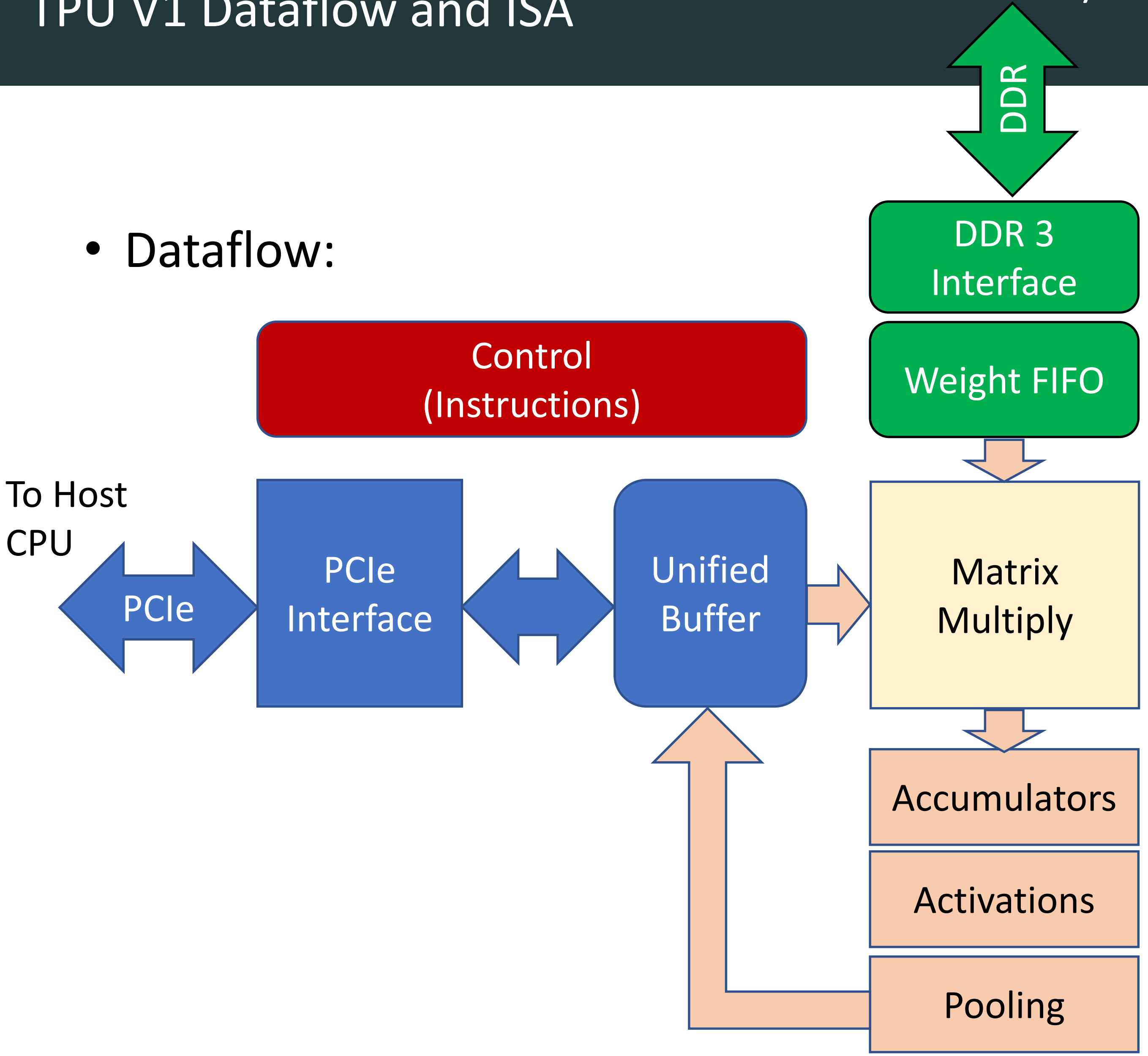
TPU Data Rates



Source: https://cloud.google.com/blog/products/ai-machine-learning/an-in-depth-look-at-googles-first-tensor-processing-unit-tpu?hl=en

TPU V1 Dataflow and ISA

- Dataflow:

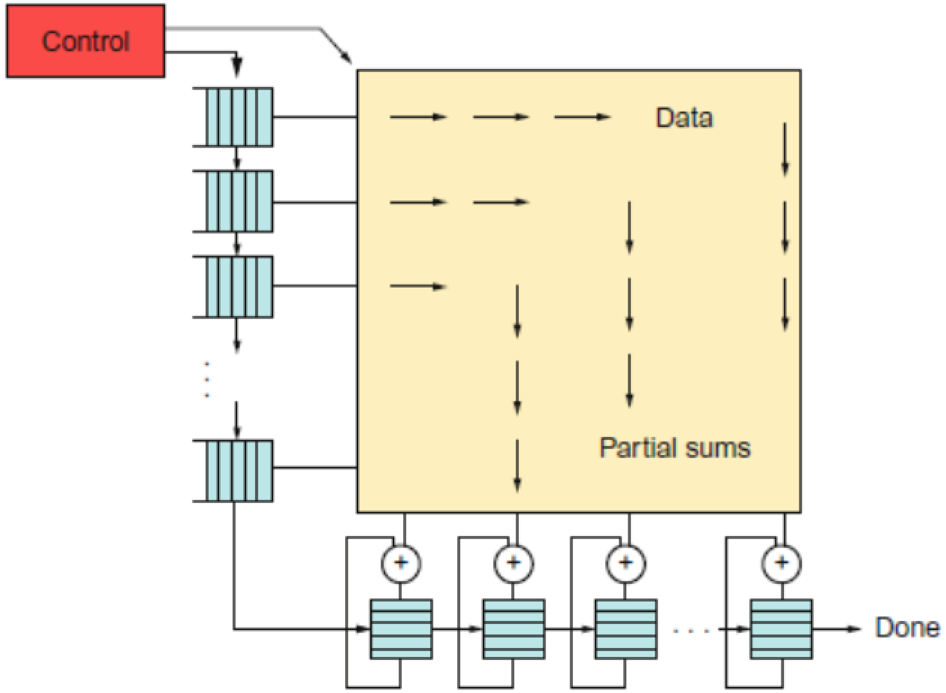


Instructions:

- Read Weights 🟩
  - Reads weights from the DDR into the Weight FIFO
- Read from Host Memory: 🟦
  - Reads data from the CPU (Host) memory into the unified TPU buffer
- Execute Matrix Matrix Multiply for Convolution + Activation + Pooling 🟨 🟧
- Write to Host Memory 🟦
  - Writes data from unified buffer into CPU memory

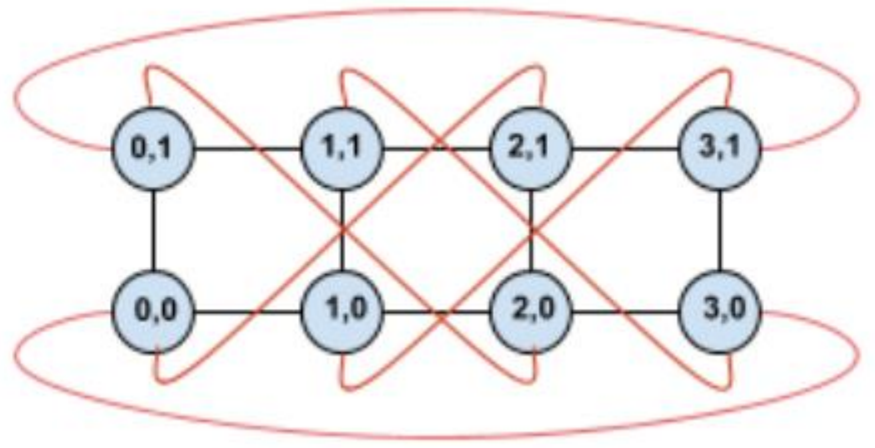
TPU: Matrix Matrix Multiply

- Core of the TPU is matrix-matrix-multiply
- 2D Systolic Array:
  - Input 1: Matrix size Sx256 (Unified buffer)
  - Input 2: Constant matrix 256x256 (Weight FIFO)
  - Output: Input1 multiplied Input 2
  - Latency: S cycles
  - Initialization interval: 1

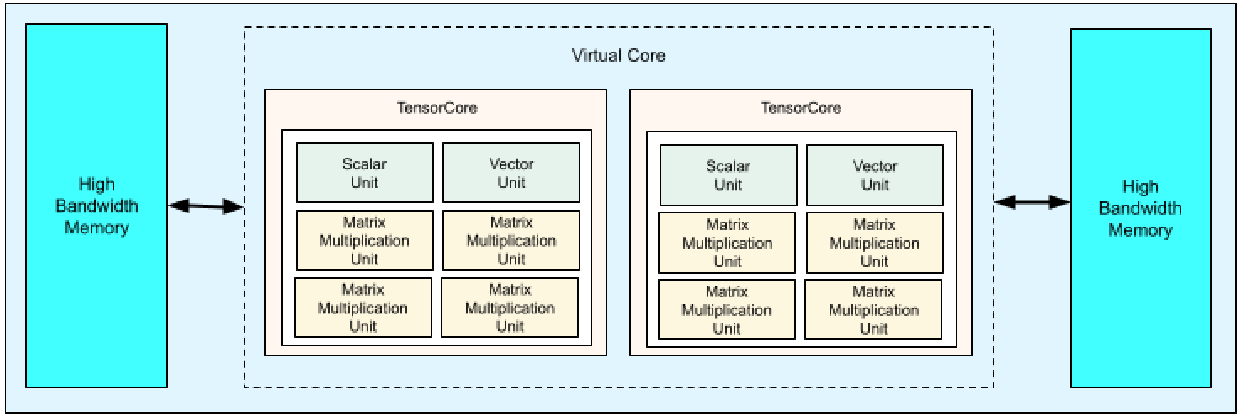


Google TPU V4 for Cloud

Key specifications	v4 Pod values
Peak compute per chip	275 teraflops (bf16 or int8)
HBM2 capacity and bandwidth	32 GiB, 1200 GBps
Measured min/mean/max power	90/170/192 W
TPU Pod size	4096 chips
Interconnect topology	3D mesh
Peak compute per Pod	1.1 exaflops (bf16 or int8)
All-reduce bandwidth per Pod	1.1 PB/s
Bisection bandwidth per Pod	24 TB/s

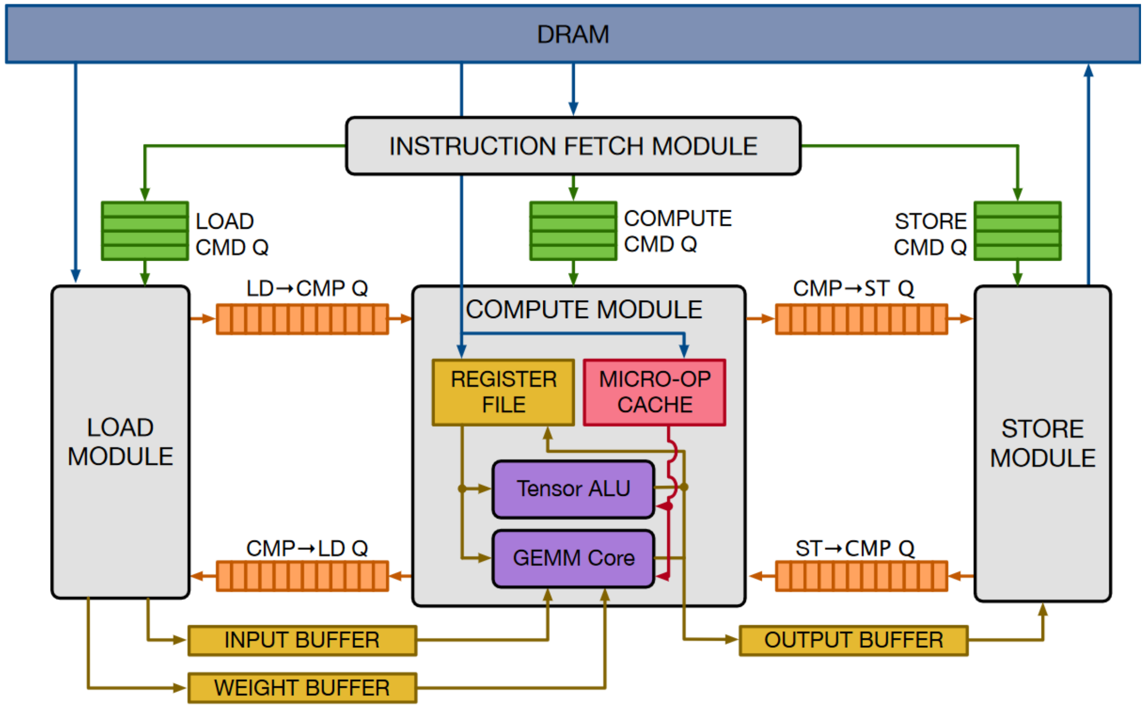


Chips can be arranged in Twisted Torus interconnect


High Bandwidth Memory
Virtual Core
TensorCore
Scalar Unit
Vector Unit
Matrix Multiplication Unit
Matrix Multiplication Unit
Matrix Multiplication Unit
Matrix Multiplication Unit
TensorCore
Scalar Unit
Vector Unit
Matrix Multiplication Unit
Matrix Multiplication Unit
Matrix Multiplication Unit
Matrix Multiplication Unit
High Bandwidth Memory

Source: https://cloud.google.com/tpu/docs/v4

Embedded NPU: Versatile Tensor Accelerator (VTA)



- Source: http://arxiv.org/pdf/1807.04188
- Open Source: https://github.com/apache/tvm-vta

Summary

Covered Topics

- General-Purpose Processor Cores
  - Pipelining
  - Speculation and Branch Prediction
  - Instruction-Level Parallelism: Superscalar, VLIW
  - Thread-Level Parallelism: Multi-threading, Multi-Core
  - Data-Level Parallelism: Vector

- Specialized Cores :
  - GP-GPUs
  - Accelerators: TPU, NPU

Thank you for your attention!